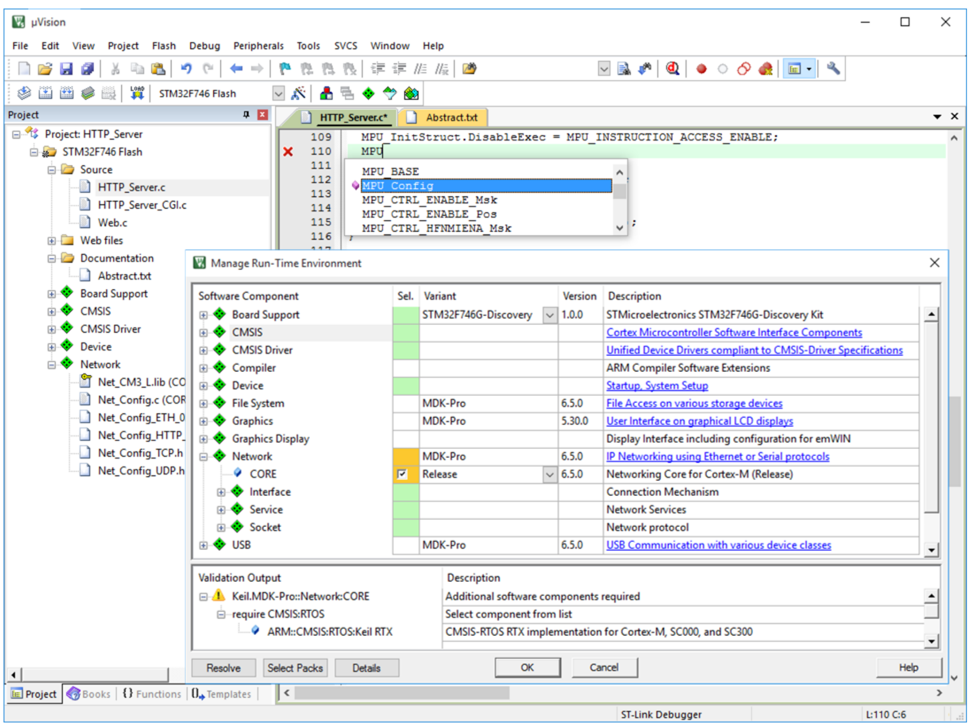
ARM® KEIL®

Microcontroller Tools

Getting Started with MDK

Create Applications with µVision®
for ARM® Cortex®-M Microcontrollers



NOTE
We assume you are familiar with Microsoft Windows, the hardware, and the instruction set of the ARM® Cortex®-M processor.

Preface

Thank you for using the MDK Version 5 Microcontroller Development Kit available from ARM® Keil®. To provide you with the very best software tools for developing ARM® Cortex®-M processor based embedded applications we design our tools to make software engineering easy and productive. ARM also offers complementary products such as the ULINK™ debug and trace adapters and a range of evaluation boards. MDK is expandable with various third party tools, starter kits, and debug adapters.

Chapter Overview

The book starts with the installation of MDK and describes the software components along with complete workflow from starting a project up to debugging on hardware. It contains the following chapters:

MDK Introduction provides an overview about the MDK Tools, the Software Packs, and describes the product installation along with the use of example projects.

CMSIS is a software framework for embedded applications that run on Cortex-M based microcontrollers. It provides consistent software interfaces and hardware abstraction layers that simplify software reuse.

Software Components enable retargeting of I/O functions for various standard I/O channels and add board support for a wide range of evaluation boards.

Create Applications guides you towards creating and modifying projects using CMSIS and device-related software components. A hands-on tutorial shows the main configuration dialogs for setting tool options.

Debug Applications describes the process of debugging applications on real hardware and explains how to connect to development boards using a wide range of debug adapters.

Middleware gives further details on the middleware that is available for users of the MDK-Professional and MDK-Plus editions.

Using Middleware explains how to create applications that use the middleware available with MDK-Professional and MDK-Plus and contains essential tips and tricks to get you started quickly.

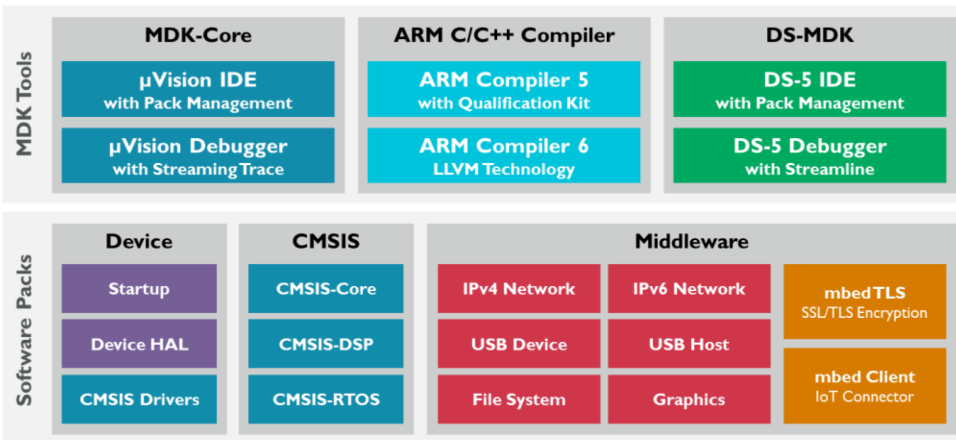
Contents

NOTE
This user's guide describes how to create projects for ARM Cortex-M microcontrollers using the µVision IDE/Debugger.

Refer to the ***Getting Started with DS-MDK*** *user's guide for information how to create applications with the Eclipse-based DS-5 IDE/Debugger for ARM Cortex-A/Cortex-M devices.*

MDK Introduction

The Keil Microcontroller Development Kit (MDK) helps you to create embedded applications for ARM Cortex-M processor-based devices. MDK is a powerful, yet easy to learn and use development system. MDK consists of the MDK Core plus device-specific Software Packs, which can be downloaded and installed based on the requirements of your application.



MDK Tools

The MDK Tools include all the components that you need to create, build, and debug an embedded application for ARM based microcontroller devices. The **MDK-Core** is based on the genuine Keil µVision IDE/Debugger with leading support for Cortex-M processor-based microcontroller devices including the new ARMv8-M architecture. **DS-MDK** contains the Eclipse-based DS-5 IDE/Debugger and offers multi-processor support for devices based on 32-bit Cortex-A processors or hybrid systems with 32-bit Cortex-A and Cortex-M processors.

MDK includes two **ARM C/C++ Compilers** with assembler, linker, and highly optimize run-time libraries tailored for optimum code size and performance:

- ARM Compiler Version 5 is the reference C/C++ compiler available with a TÜV certified Qualification Kit and Long-Term Support and Maintenance.
- ARM Compiler Version 6 is based on the innovative LLVM technology and supports the latest C language standards including C++11 and C++14.

Software Packs

Software Packs contain device support, CMSIS libraries, middleware, board support, code templates, and example projects. They may be added any time to MDK Core or DS-MDK, making new device support and middleware updates independent from the toolchain. The IDE manages the provided software components that are available for the application as building blocks.

MDK Editions

The product selector, available at **http://www.keil.com/mdk5/selector**, gives an overview of the features enabled in each edition:

- **MDK-Professional** contains all features of **MDK-Plus**. In addition, it supports IPv4/IPv6 dual-stack networking, IoT connectivity, and a USB Host stack. Once available, MDK-Professional includes ARMv8-M architecture support and a license for DS-MDK.
- **MDK-Plus** contains middleware libraries for IPv4 networking, USB Device, File System, and Graphics. It supports ARM Cortex-M, selected ARM Cortex-R, ARM7, and ARM9 processor based microcontrollers.
- **MDK-Cortex-M** supports Cortex-M processor-based microcontrollers.
- **MDK-Lite** is code size restricted to 32 KByte and intended for product evaluation, small projects, and the educational market.

License Types

With the exception of **MDK-Lite**, the MDK editions require activation using a license code. The following licenses types are available:

Single-User License (Node-Locked) grants the right to use the product by one developer on two computers at the same time.

Floating-User License or **FlexLM License** grants the right to use the product on several computers by a number of developers at the same time.

7-Day MDK-Professional Trial License to test the comprehensive middleware without code size limits.

For further details, refer to the *Licensing User's Guide* at **www.keil.com/support/man/docs/license**.

Installation

Software and Hardware Requirements

MDK has the following minimum hardware and software requirements:

A PC running Microsoft Windows (32-bit or 64-bit) operating system

4 GB RAM and 8 GB hard-disk space

1280 x 800 or higher screen resolution; a mouse or other pointing device

Install MDK Core

Download **MDK Version 5** from **www.keil.com/download** **-** Product Downloads and run the installer.

Follow the instructions to install the MDK Core on your local computer. The installation also adds the Software Packs for ARM **CMSIS** and MDK **Middleware**.

MDK Version 5 is capable of using MDK Version 4 projects after installation of the Legacy Support from **www.keil.com/mdk5/legacy**. This adds support for ARM7, ARM9, and Cortex-R processor-based devices.

After the MDK Core installation is complete, the **Pack Installer** is started automatically, which allows you to add supplementary Software Packs. As a minimum, you need to install a Software Pack that supports your target microcontroller device.

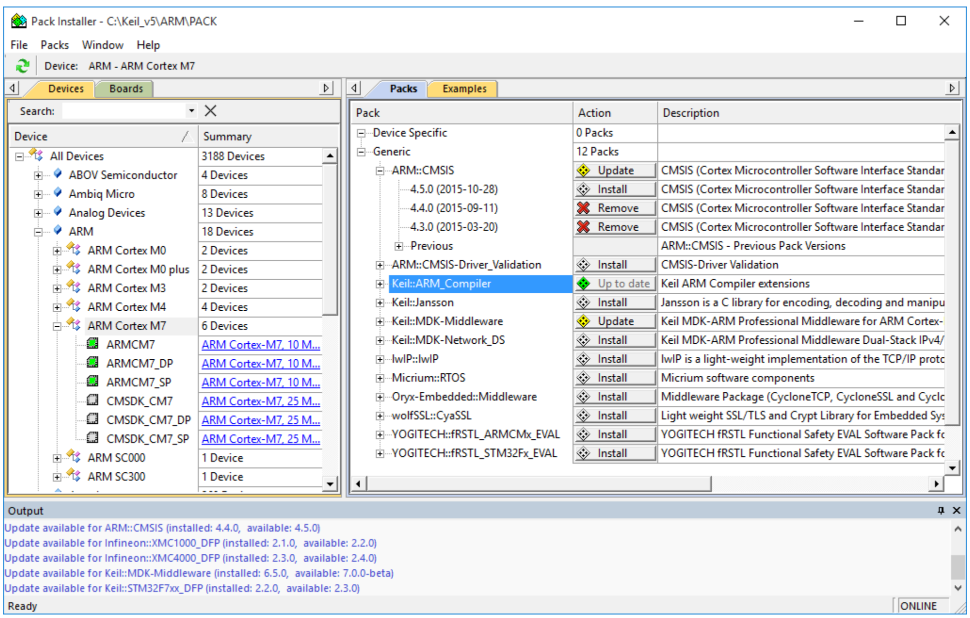
Install Software Packs

The **Pack Installer** is a utility for managing Software Packs on the local computer.

The **Pack Installer** runs automatically during the installation, but also can be run from µVision using the menu item **Project – Manage – Pack Installer**. To get access to devices and example projects you should install the Software Pack related to your target device or evaluation board.

NOTE

To obtain information of published Software Packs the Pack Installer connects to ***www.keil.com/pack***.



The status bar, located at the bottom of the Pack Installer, shows information about the Internet connection and the installation progress.

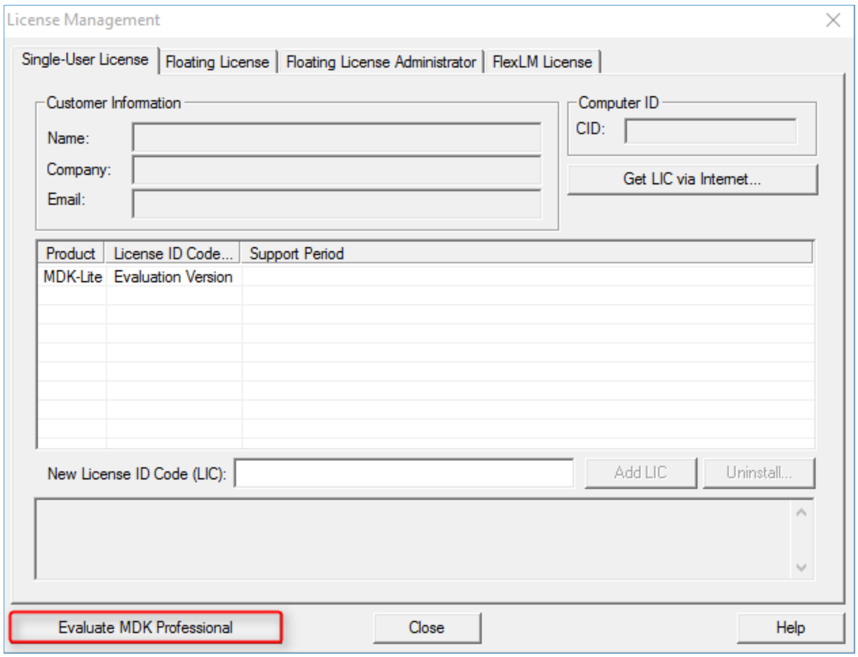
TIP: The device database at **www.keil.com/dd2** lists all available devices and provides download access to the related Software Packs. If the Pack Installer cannot access **www.keil.com/pack** you can manually install Software Packs using the menu command **File – Import** or by double-clicking *.PACK files.

MDK-Professional Trial License

MDK has a built-in **free** seven-day trial license for MDK-Professional. This removes the code size limits and you can explore and test the comprehensive middleware.

Start µVision with administration rights.

1. In µVision, go to **File – License Management...** and click **Evaluate MDK Professional**



2. On the next screen, click **Start MDK Professional Evaluation for 7 Days**. After the installation, the screen displays information about the expiration date and time.

NOTE
*Activation of the 7-day MDK Professional trial version enables the option **Use Flex Server** in the tab **FlexLM License** as this license is based on FlexLM.*

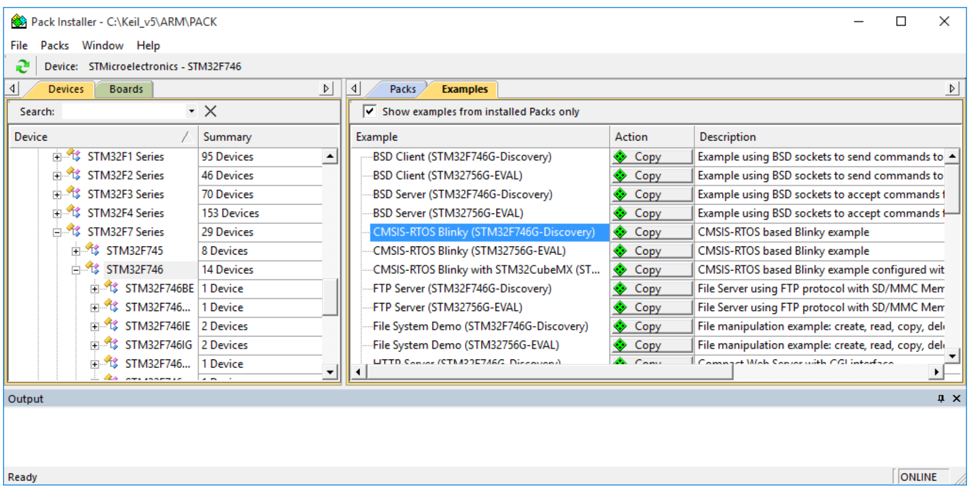
Verify Installation using Example Projects

Once you have selected, downloaded, and installed a Software Pack for your device, you can verify your installation using one of the examples provided in the Software Pack. To verify the Software Pack installation, we recommend using a *Blinky* example, which typically flashes LEDs on a target board.

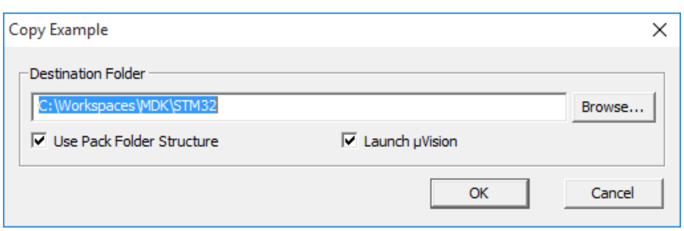
TIP: Review the getting started video on **http://www.keil.com/mdk5** that explains how to connect and work with an evaluation kit.

Copy an Example Project

In the Pack Installer, select the tab **Examples**. Use filters in the toolbar to narrow the list of examples.



Click **Copy** and enter the **Destination Folder** name of your working directory.



NOTE

You must copy the example projects to a working directory of your choice.

Enable **Launch µVision** to open the example project directly in the IDE.

Enable **Use Pack Folder Structure** to copy example projects into a common folder. This avoids overwriting files from other example projects. Disable **Use Pack Folder Structure** to reduce the complexity of the example path.

Click **OK** to start the copy process.

Use an Example Application with µVision

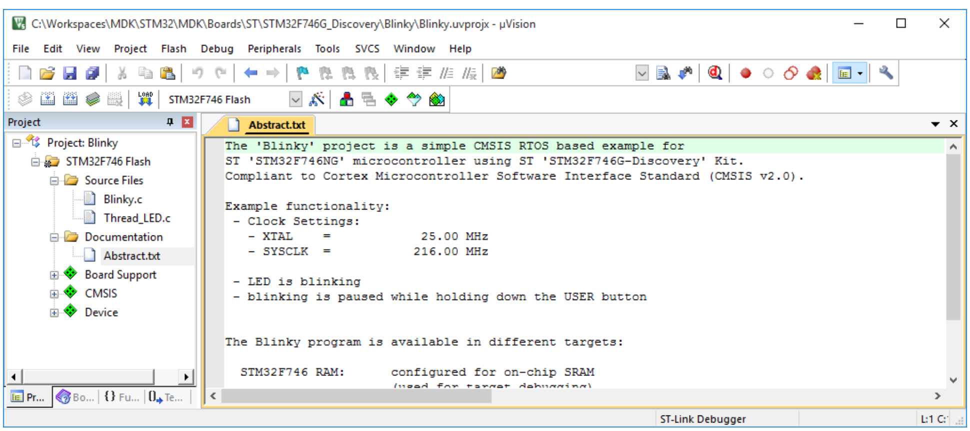
Now µVision starts and loads the example project where you can:

 Build the application, which compiles and links the related source files.

 Download the application, typically to on-chip Flash ROM of a device.

 Run the application on the target hardware using a debugger.

The step-by-step instructions show you how to execute these tasks. After copying the example, µVision starts and looks similar to the picture below.

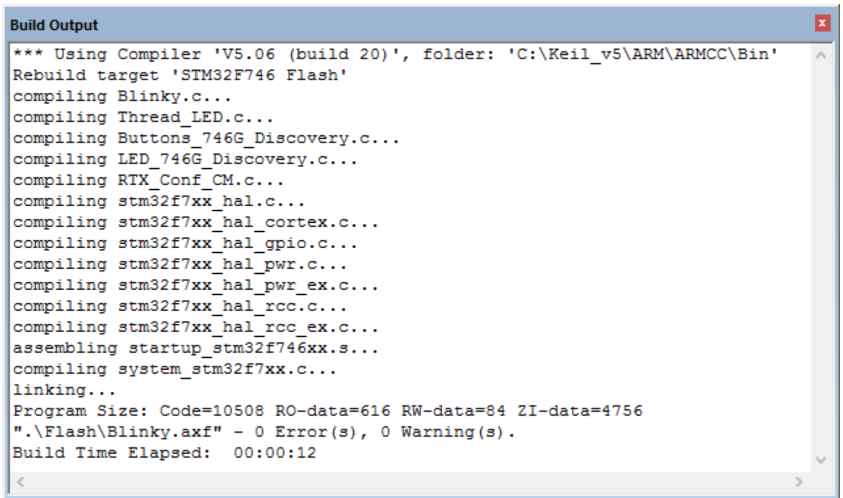


TIP: Most example projects contain an *Abstract.txt* file with essential information about the operation and hardware configuration.

Build the Application

Build the application using the toolbar button **Rebuild.**

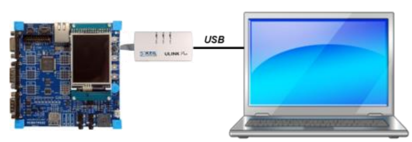
The **Build Output** window shows information about the build process. An error-free build shows information about the program size.




```
*** Using Compiler 'V5.06 (build 20)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Rebuild target 'STM32F746 Flash'
compiling Blinky.c...
compiling Thread_LED.c...
compiling Buttons_746G_Discovery.c...
compiling LED_746G_Discovery.c...
compiling RTX_Conf_CM.c...
compiling stm32f7xx_hal.c...
compiling stm32f7xx_hal_cortex.c...
compiling stm32f7xx_hal_gpio.c...
compiling stm32f7xx_hal_pwr.c...
compiling stm32f7xx_hal_pwr_ex.c...
compiling stm32f7xx_hal_rcc.c...
compiling stm32f7xx_hal_rcc_ex.c...
assembling startup_stm32f746xx.s...
compiling system_stm32f7xx.c...
linking...
Program Size: Code=10508 RO-data=616 RW-data=84 ZI-data=4756
".\Flash\Blinky.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:12
```

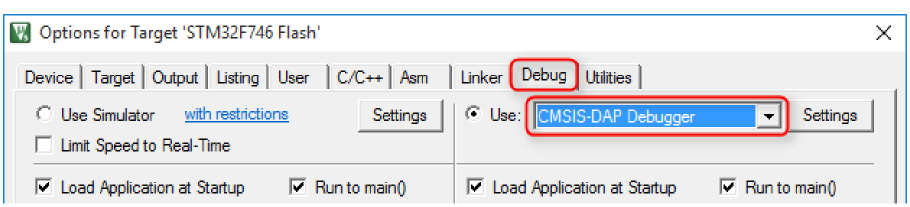
Download the Application

Connect the target hardware to your computer using a *debug adapter* that typically connects via USB. Several evaluation boards provide an on-board debug adapter.



Now, review the settings for the debug adapter. Typically, example projects are pre-configured for evaluation kits; thus, you do not need to modify these settings.

Click **Options for Target** on the toolbar and select the **Debug** tab. Verify that the correct debug adapter of the evaluation board you are using is selected and enabled. For example, **CMSIS-DAP Debugger** is a debug adapter that is part of several starter kits.

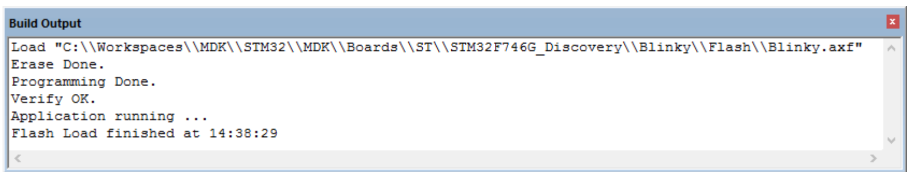


Enable **Load Application at Startup** for loading the application into the µVision Debugger whenever a debugging session is started.

Enable **Run to main()** for executing the instructions up to the first executable statement of the main() function. The instructions are executed upon each RESET.

TIP: Click the button **Settings** to verify communication settings and diagnose problems with your target hardware. For further details, click the button **Help** in the dialogs. If you have any problems, refer to the user guide of the starter kit.

Click **Download** on the toolbar to load the application to your target hardware.



The **Build Output** window shows information about the download progress.

Run the Application

Click **Start/Stop Debug Session** on the toolbar to start debugging the application on hardware.

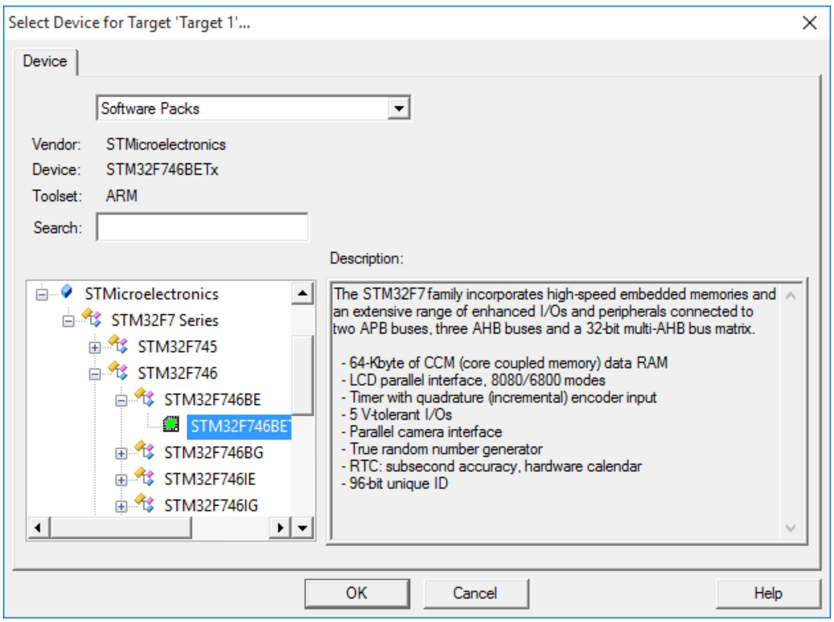
Click **Run** on the debug toolbar to start executing the application. LEDs should flash on the target hardware.

Use Software Packs

Software Packs contain information about microcontroller devices and software components that are available for the application as building blocks.

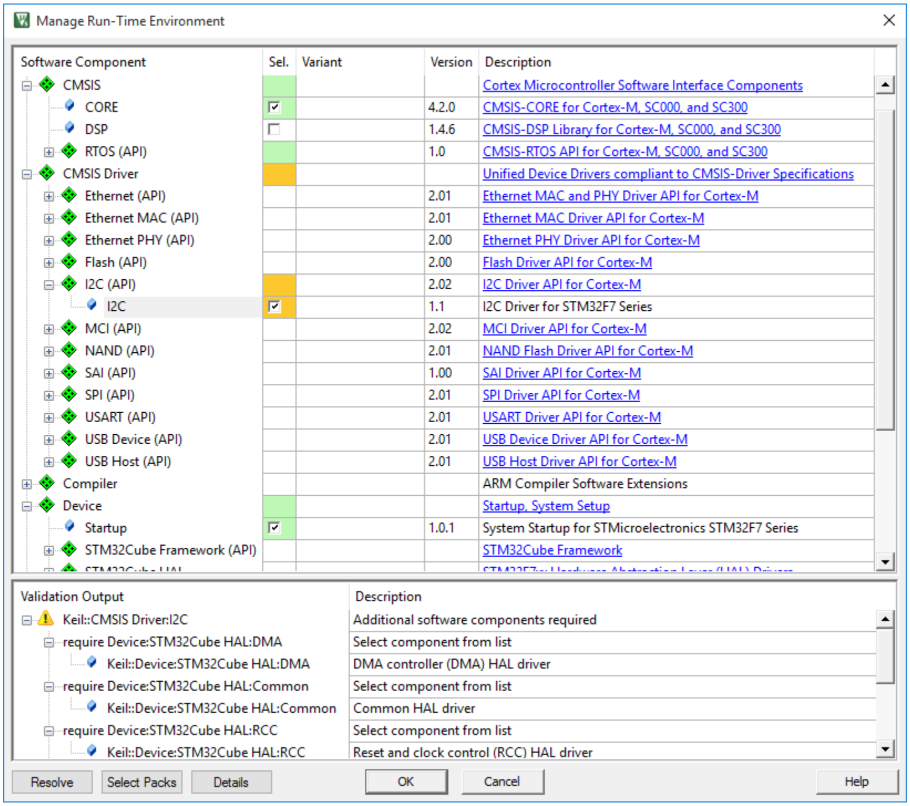
The device information pre-configures development tools for you and shows only the options that are relevant for the selected device.

Start µVision and use the menu **Project - New µVision Project**. After you have selected a project directory and specified the project name, select a target device.



TIP: Only devices that are part of the installed Software Packs are shown. If you are missing a device, use the Pack Installer to add the related Software Pack. The search box helps you to narrow down the list of devices.

After selecting the device, the **Manage Run-Time Environment** window shows the related software components for this device.



TIP: The links in the column *Description* provide access to the documentation of each software component.

NOTE

*The notation **::<Component Class>:<Group>:<Name>** is used to refer to components. For example, **::CMSIS:CORE** refers to the component CMSIS-CORE selected in the dialog above.*

Software Component Overview

The following table shows the software components for a typical installation. Depending on your selected device, some of these software components might not be visible in the Manage Run-Time Environment window. In case you have installed additional Software Packs, more software components will be available.

Software Component	Description	Page
Board Support	Interfaces to the peripherals of evaluation boards.	44
CMSIS	CMSIS interface components, such as CORE, DSP, and CMSIS-RTOS.	22
CMSIS Driver	Unified device drivers for middleware and user applications.	39
Compiler	ARM Compiler specific software components to retarget I/O operations for example for printf style debugging.	42
Device	System startup and low-level device drivers.	46
File System	Middleware component for file access on various storage device types.	84
Graphics	Middleware component for creating graphical user interfaces.	86
Network	Middleware component for TCP/IP networking using Ethernet or serial protocols.	82
USB	Middleware component for USB Host and USB Device supporting standard USB Device classes.	85

Product Lifecycle Management with Software Packs

MDK allows you to install multiple versions of a Software Pack. This enables Product Lifecycle Management (PLM) as it is common for many projects.

There are four distinct phases of PLM:

Concept: Definition of major project requirements and exploration with a functional prototype.

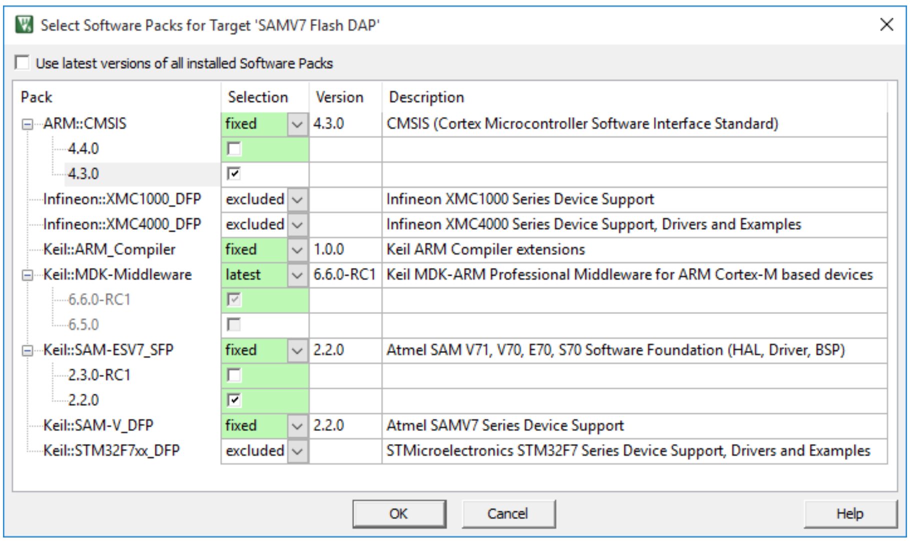
Design: Prototype testing and implementation of the product based on the final technical features and requirements.

Release: The product is manufactured and brought to market.

Service: Maintenance of the products including support for customers; finally phase-out or end-of-life.

In the concept and design phase, you normally want to use the latest Software Packs to be able to incorporate new features and bug fixes quickly. Before product release, you will freeze the Software Components to a known tested state. In the product service phase, use the fixed versions of the Software Components to support customers in the field.

The dialog **Select Software Packs** helps you to manage the versions of each Software Pack in your project:



When the project is completed, disable the option **Use latest version of all installed Software Packs** and specify the Software Packs with the settings under **Selection**:

latest: use the latest version of a Software Pack. Software Components are updated when a newer Software Pack version is installed.

fixed: specify an installed version of the Software Pack. Software Components in the project target will use these versions.

excluded: no Software Components from this Software Pack are used.

The colors indicate the usage of Software Components in the current project target:

Some Software Components from this Pack are used.

Some Software Components from this Pack are used, but the Pack is excluded.

No Software Component from this Pack is used.

Software Version Control Systems (SVCS)

µVision carries template files for GIT, SVN, CVS, and others to support Software Version Control Systems (SVCS).

Application note 279 "Using Git for Project Management with µVision" (**www.keil.com/appnotes/docs/apnt_279.asp**) describes how to establish a robust workflow for version control of projects using Software Packs.

Access Documentation

MDK provides online manuals and context-sensitive help. The µVision **Help** menu opens the main help system that includes the *µVision User's Guide*, getting started manuals, compiler, linker and assembler reference guides.

Many dialogs have context-sensitive **Help** buttons that access the documentation and explain dialog options and settings.

You can press **F1** in the editor to access help on language elements like RTOS functions, compiler directives, or library routines. Use **F1** in the command line of the **Output** window for help on debug commands, and some error and warning messages.

The **Books** window may include device reference guides, data sheets, or board manuals. You can even add your own documentation and enable it in the **Books** window using the menu **Project – Manage – Components, Environment, Books – Books**.

The **Manage Run-Time Environment** dialog offers access to documentation via links in the *Description* column.

In the **Project** window, you can right-click a software component group and open the documentation of the corresponding element.

You can access the **µVision User's Guide** on-line at **www.keil.com/support/man/docs/uv4**.

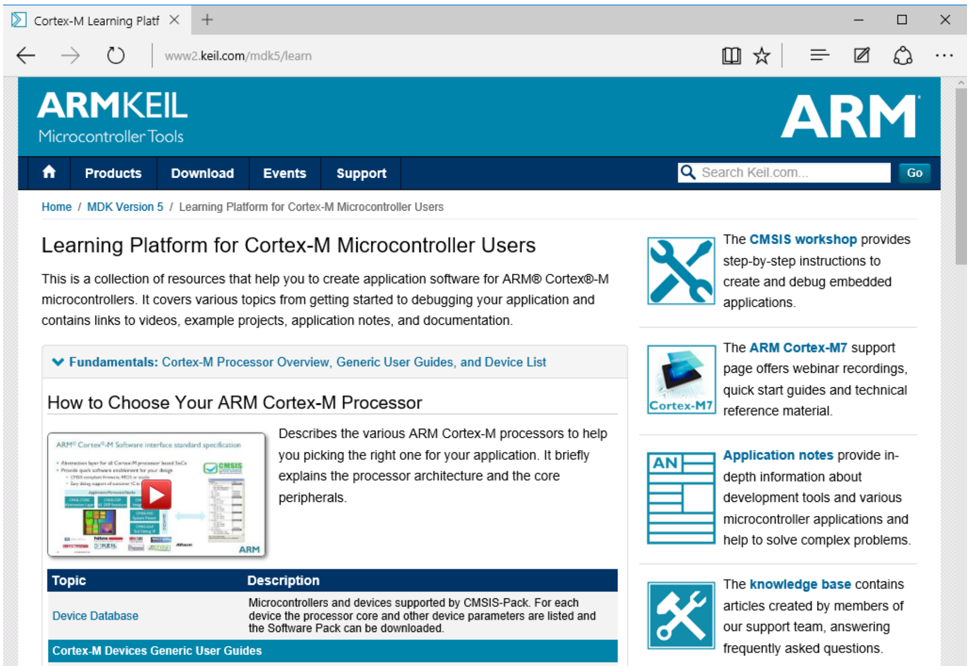
Request Assistance

If you have suggestions or you have discovered an issue with the software, please report them to us. Support and information channels are accessible at **www.keil.com/support**.

When reporting an issue, include your license code (if you have one) and product version, available from the µVision menu **Help – About**.

Learning Platform

We offer a website that helps you to learn more about the programming of ARM Cortex-based microcontrollers. It contains tutorials, videos, further documentation, as well as useful links to other websites and is available at www.keil.com/learn.



Quick Start Guides

Quick Start Guides help you to bring up your target hardware quickly. They describe the required steps to get a development board up and running with MDK and list required Software Packs as well as driver requirements for integrated debug adapters.

NOTE

www.keil.com/mdk5/qsg *explains how to download the quick start guides*

CMSIS

The **Cortex Microcontroller Software Interface Standard** (CMSIS) provides a ground-up software framework for embedded applications that run on Cortex-M based microcontrollers. CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software reuse, reducing the learning curve for microcontroller developers.

NOTE
This chapter is a reference section. The chapter *Create Applications on page 45* *shows you how to use CMSIS for creating application code.*

The CMSIS, defined in close cooperation with various silicon and software vendors, provides a common approach to interface peripherals, real-time operating systems, and middleware components.

The CMSIS application software components are:

- **CMSIS-CORE**: Defines the API for the Cortex-M processor core and peripherals and includes a consistent system startup code. The software components **::CMSIS:CORE** and **::Device:Startup** are all you need to create and run applications on the native processor that uses exceptions, interrupts, and device peripherals.
- **CMSIS-RTOS RTX:** Provides a standardized real-time operating system API and enables software templates, middleware, libraries, and other components that can work across supported RTOS systems. This manual explains the usage of the CMSIS-RTOS RTX implementation.
- **CMSIS-DSP**: Is a library collection for digital signal processing (DSP) with over 60 Functions for various data types: fix-point (fractional q7, q15, q31) and single precision floating-point (32-bit).
- **CMSIS-Driver**: Is a software API that describes peripheral driver interfaces for middleware stacks and user applications. The CMSIS-Driver API is designed to be generic and independent of a specific RTOS making it reusable across a wide range of supported microcontroller devices.

CMSIS-CORE

This section explains the usage of CMSIS-CORE in applications that run natively on a Cortex-M processor. This type of operation is known as *bare-metal*, because it uses no real-time operating system.

Using CMSIS-CORE

A native Cortex-M application with CMSIS uses the software component **::CMSIS:CORE**, which should be used together with the software component **::Device:Startup**. These components provide the following central files:

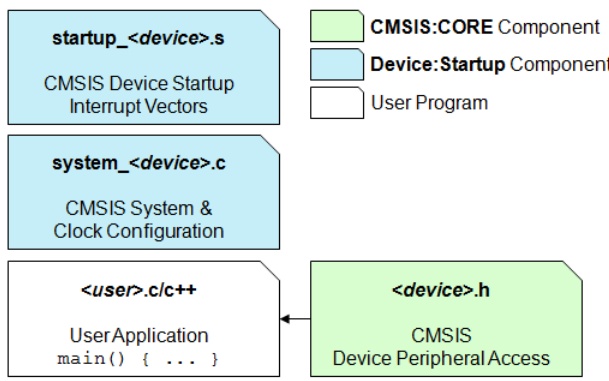
NOTE

In actual file names, <device> is the name of the microcontroller device.

The *startup_<device>.s* file with reset handler and exception vectors.

The *system_<device>.c* configuration file for basic device setup (clock and memory BUS).

The *<device>.h* include file for user code access to the microcontroller device.



The *<device>.h* header file is included in C source files and defines:

Peripheral access with standardized register layout.

Access to interrupts and exceptions, and the Nested Interrupt Vector Controller (NVIC).

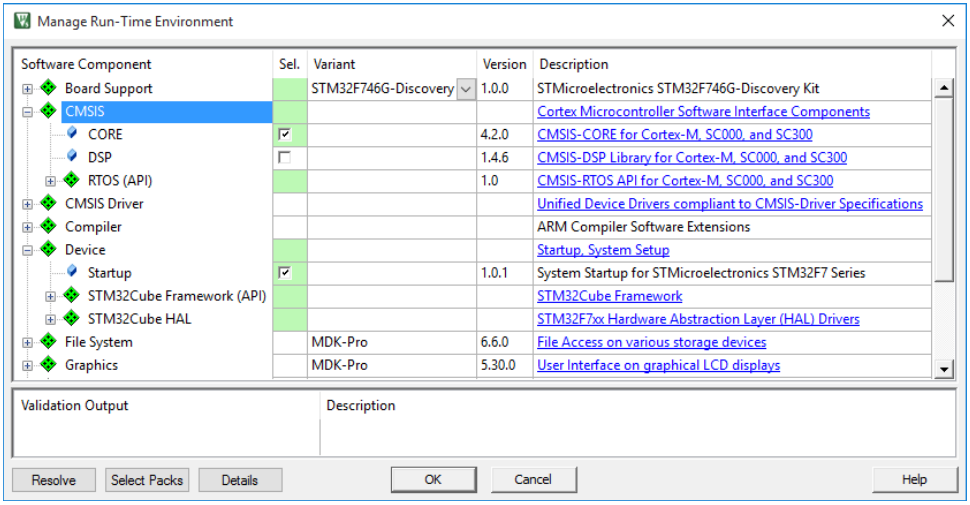
Intrinsic functions to generate special instructions, for example to activate sleep mode.

Systick timer (SYSTICK) functions to configure and start a periodic timer interrupt.

Debug access for *printf*-style I/O and ITM communication via on-chip CoreSight™.

Adding Software Components to the Project

The files for the components **::CMSIS:CORE** and **::Device:Startup** are added to a project using the µVision dialog **Manage Run-Time Environment**. Just select the software components as shown below:



The µVision environment adds the related files.

Source Code Example

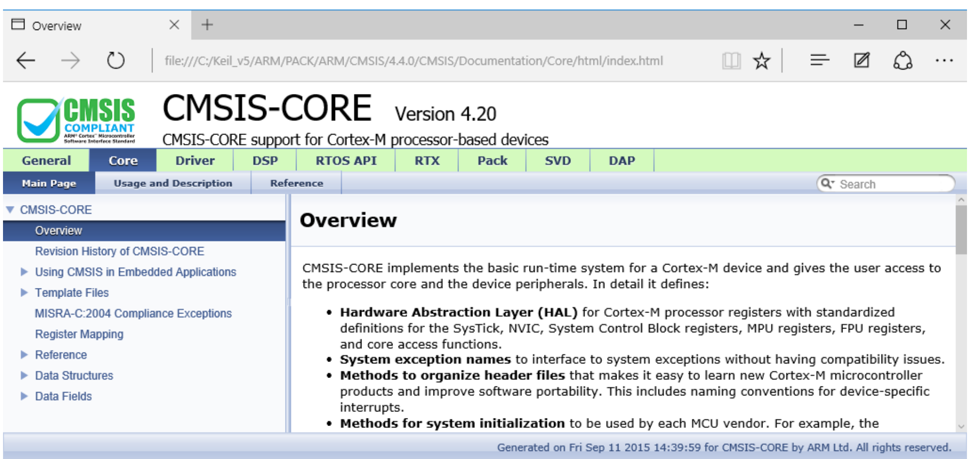
The following source code lines show the usage of the CMSIS-CORE layer.

Example of using the CMSIS-CORE layer

```
#include "stm32f4xx.h"                  // File name depends on device used

uint32_t volatile msTicks;              // Counter for millisecond Interval
uint32_t volatile frequency;            // Frequency for timer


void SysTick_Handler (void) {           // SysTick Interrupt Handler
  msTicks++;                            // Increment Counter
}

void WaitForTick (void) {
  uint32_t curTicks;
  curTicks = msTicks;                   // Save Current SysTick Value
  while (msTicks == curTicks) {         // Wait for next SysTick Interrupt
    __WFE ();                           // Power-Down until next Event
  }
}

void TIM1_UP_IRQHandler (void) {        // Timer Interrupt Handler
  ; // Add user code here
}
```

```
void timer1_init(int frequency) {      // Set up Timer (device specific)
  NVIC_SetPriority (TIM1_UP_IRQn, 1); // Set Timer priority
  NVIC_EnableIRQ (TIM1_UP_IRQn);      // Enable Timer Interrupt
}


// Configure & Initialize the MCU
void Device_Initialization (void) {
  if (SysTick_Config (SystemCoreClock / 1000)) {     // SysTick 1ms
    : // Handle Error
  }
  timer1_init (frequency);        // Setup device-specific timer
}

// The processor clock is initialized by CMSIS startup + system file
int main (void) {                 // User application starts here
  Device_Initialization ();       // Configure & Initialize MCU

  while (1) {                     // Endless Loop (the Super-Loop)
    __disable_irq ();             // Disable all interrupts
    // Get_InputValues ();
    __enable_irq ();              // Enable all interrupts
    // Process_Values ();
    WaitForTick ();               // Synchronize to SysTick Timer
  }
}
```

For more information, right-click the group CMSIS in the Project window, and choose **Open Documentation**, or refer to the CMSIS-CORE documentation **http://www.keil.com/cmsis/core**.



CMSIS-RTOS RTX

This section introduces the CMSIS-RTOS RTX Real-Time Operating System, describes its features and advantages, and explains configuration settings of this RTOS.

NOTE
MDK is compatible with many third-party RTOS solutions. However, CMSIS-RTOS RTX is well integrated into MDK, is feature-rich and tailored towards the requirements of deeply embedded systems.

Software Concepts

There are two basic design concepts for embedded applications:

Infinite Loop Design: involves running the program as an endless loop. Program functions (threads) are called from within the loop, while interrupt service routines (ISRs) perform time-critical jobs including some data processing.

RTOS Design: involves running several threads with a **Real-Time Operating System (RTOS)**. The RTOS provides inter-thread communication and time management functions. A preemptive RTOS reduces the complexity of interrupt functions, because high-priority threads can perform time-critical data processing.

Infinite Loop Design

Running an embedded program in an endless loop is an adequate solution for simple embedded applications. Time-critical functions, typically triggered by hardware interrupts, execute in an ISR that also performs any required data processing. The main loop contains only basic operations that are not time-critical and run in the background.

Advantages of an RTOS Kernel

RTOS kernels, like the CMSIS-RTOS RTX, are based on the idea of parallel execution threads (tasks). As in the real world, your application will have to fulfill multiple different tasks. An RTOS-based application recreates this model in your software with various benefits:

Thread priority and run-time scheduling is handled by the RTOS Kernel, using a proven code base.

The RTOS provides a well-defined interface for communication between threads.

A pre-emptive multi-tasking concept simplifies the progressive enhancement of an application even across a larger development team. New functionality can be added without risking the response time of more critical threads.

Infinite loop software concepts often poll for occurred interrupts. In contrast, RTOS kernels themselves are interrupt driven and can largely eliminate polling. This allows the CPU to sleep or process threads more often.

Modern RTOS kernels are transparent to the interrupt system, which is mandatory for systems with hard real-time requirements. Communication facilities can be used for IRQ-to-task communication and allow top-half/bottom-half handling of your interrupts.

Using CMSIS-RTOS RTX

CMSIS-RTOS RTX is implemented as a library and exposes the functionality through the header file *cmsis_os.h*.

Execution of the CMSIS-RTOS RTX starts with the function *main()* as the first thread. This has the benefit that developers can initialize other middleware libraries that create threads internally, but the remaining part of the user application uses just the **main** thread. Consequently, the usage of the RTOS can be invisible to the application programmer, but libraries can use CMSIS-RTOS RTX features.

The software component **::CMSIS:RTOS:Keil RTX** must be used together with the components **::CMSIS:CORE** and **::Device:Startup**. Selecting these components provides the following central CMSIS-RTOS RTX files:

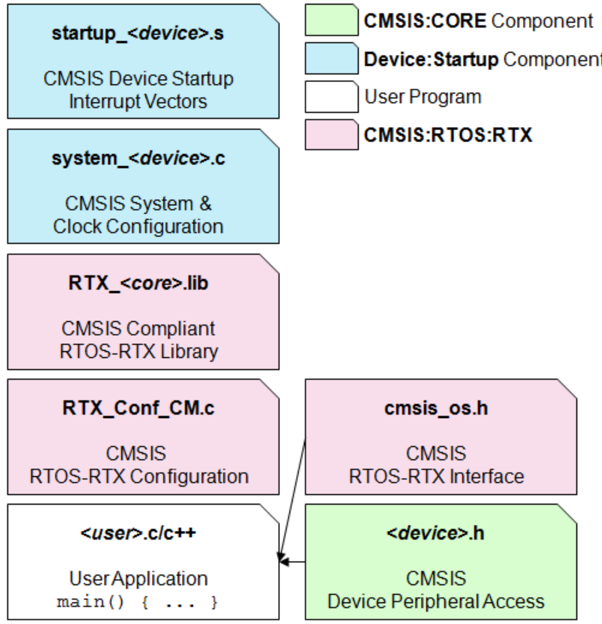
NOTE
In the actual file names, <device> is the name of the microcontroller device; <device core> represents the device processor family.

The file *RTX_<core>.lib* is the library with RTOS functions.

The configuration file *RTX_Conf_CM.c* for defining thread options, timer configurations, and RTX kernel settings.

The header file *cmsis_os.h* exposes the RTX functionality to the user application.

The function *main()* is executed as a thread.



Once these files are part of the project, developers can start using the CMSIS-RTOS RTX functions. The code example shows the use of CMSIS-RTOS RTX functions:

Example of using CMSIS-RTOS RTX functions

```
#include "cmsis_os.h"                        // CMSIS RTOS header file

void job1 (void const *argument) {           // Function 'job1'
  // execute some code
  osDelay (10);                              // Delay execution for 10ms
}

osThreadDef(job1, osPriorityLow, 1, 0);      // Define job1 as thread

int main (void) {
  osKernelInitialize ();                     // Initialize RTOS kernel
  // setup and initialize peripherals
  osThreadCreate (osThread(job1), NULL);     // Create the thread

  osKernelStart ();                          // Start kernel & job1 thread
}
```

Header File cmsis_os.h

The file *cmsis_os.h* is a template header file for the CMSIS-RTOS RTX and contains:

CMSIS-RTOS API function definitions.

Definitions for parameters and return types.

Status and priority values used by CMSIS-RTOS API functions.

Macros for defining threads and other kernel objects such as mutex, semaphores, or memory pools.

All definitions are prefixed with **os** to give a unique name space for the CMSIS-RTOS functions. Definitions that are prefixed **os_** are not be used in the application code but are local to this header file. All definitions and functions that belong to a module are grouped and have a common prefix, for example, **osThread** for threads.

Define and Reference Object Definitions

With the **#define osObjectsExternal**, objects are defined as external symbols. This allows creating a consistent header file for the entire project as shown below:

Example of a header file: osObjects.h

```
#include "cmsis_os.h"                                 // CMSIS RTOS header

extern void thread_1 (void const *argument);          // Function prototype
osThreadDef (thread_1, osPriorityLow, 1, 100);        // Thread definition

osPoolDef (MyPool, 10, long);                         // Pool definition
```

This header file, called *osObjects.h*, defines all objects when included in a C/C++ source file. When **#define osObjectsExternal** is present before the header file inclusion, the objects are defined as external symbols. Thus, a single consistent header file can be used throughout the entire project.

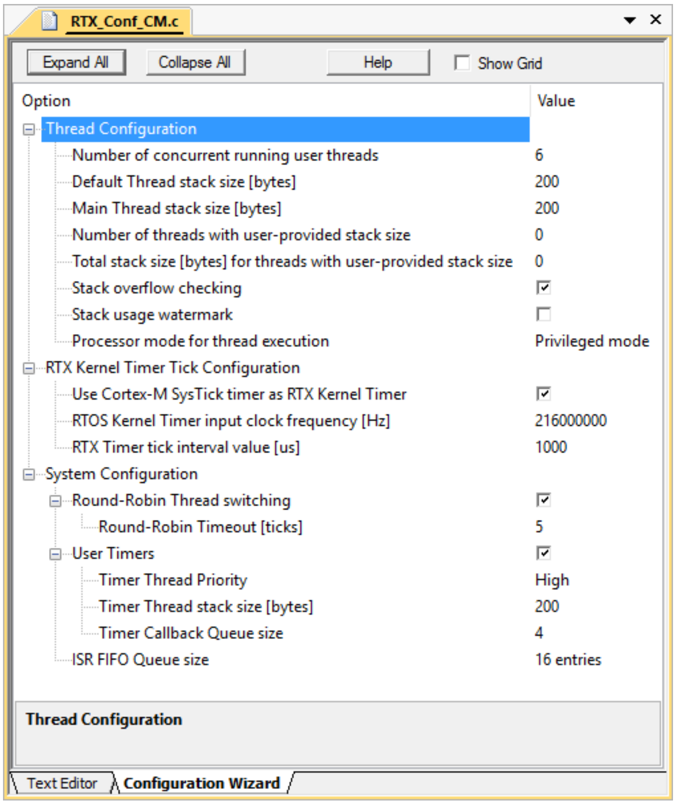
Consistent header file usage in a C file

```
#define osObjectExternal       // Objects defined as external symbols
#include "osObjects.h"         // Reference to the CMSIS-RTOS objects
```

For details, refer to the online documentation **www.keil.com/cmsis/rtos**, section **Header File Template: cmsis_os.h**.

CMSIS-RTOS RTX Configuration

The file *RTX_Conf_CM.c* contains the configuration parameters of the CMSIS-RTOS RTX. A copy of this file is part of every project using the RTX component.



You can set parameters for the thread stack, configure the Tick Timer, set Round-Robin time slice, and define user timer behaviour for threads.

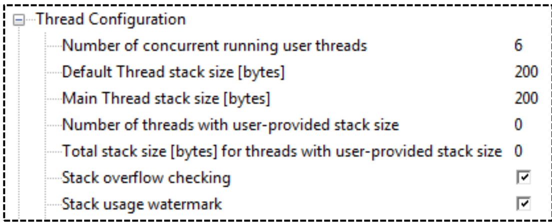
For more information about configuration options, open the RTX documentation from the **Manage Run-Time Environment** window. The section **Configuration of CMSIS-RTOS RTX** describes all available settings. The following highlights the most important settings that need adaptation in your application.

Thread Stack Configuration

Threads are defined in the code with the function *osThreadDef()*. The parameter *stacksz* specifies the stack requirement of a thread and has an impact on the method for allocating stack. CMSIS-RTOS RTX offers two methods for allocating stack requirements in the file *RTX_Conf_CM.c*:

Using a fixed memory pool: if the parameter *stacksz* is 0, then the value specified for **Default Thread stack size [bytes]** sets the stack size for the thread function.

Defining a user space: if *stacksz* is not 0, then the thread stack is allocated from a user space. The total size of this user space is specified by **Total stack size [bytes] for threads with user-provided stack size.**



Number of concurrent running threads specifies the maximum number of threads that allocate the stack from the fixed size memory pool.

Default Thread stack size [bytes] specifies the stack size (in words) for threads defined without a user-provided stack.

Main Thread stack size [bytes] is the stack requirement for the *main()* function.

Number of threads with user-provided stack size specifies the number of threads defined with a specific stack size.

Total stack size [bytes] for threads with user-provided stack size is the combined requirement (in words) of all threads defined with a specific stack size.

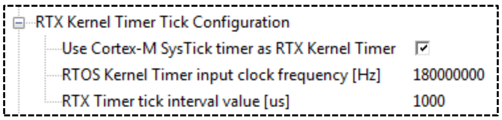
Stack overflow checking enables stack overflow check at a thread switch. Enabling this option slightly increases the execution time of a thread switch.

Stack usage watermark initializes the thread stack with a watermark pattern at the time of the thread creation. This enables monitoring of the stack usage for each thread (not only at the time of a thread switch) and helps to find stack overflow problems within a thread. Enabling this option increases significantly the execution time of *osThreadCreate()*.

NOTE
Consider these settings carefully. If you do not allocate enough memory or you do not specify enough threads, your application will not work.

RTX Kernel Timer Tick Configuration

CMSIS-RTOS RTX functions provide delays in units of milliseconds derived from the **Timer tick value**. We recommend configuring the Timer tick value to generate 1-millisecond intervals. Configuring a longer interval may reduce energy consumption, but has an impact on the granularity of the timeouts.



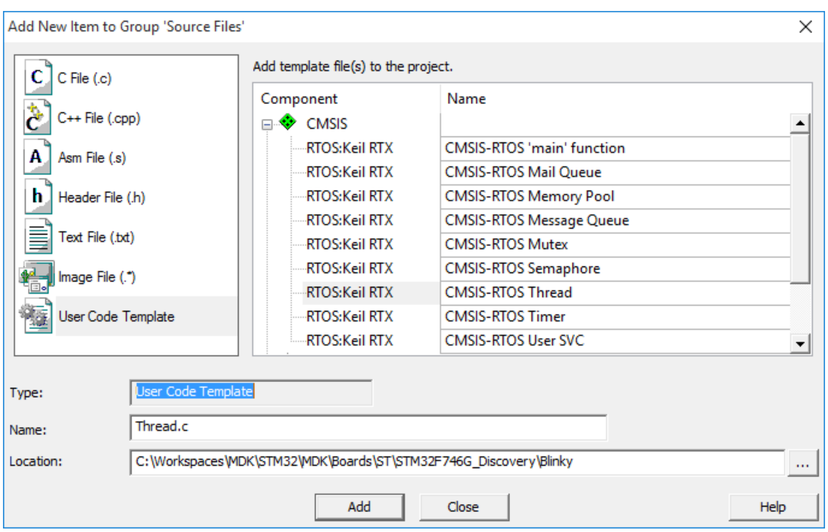
It is good practise to enable **Use Cortex-M Systick timer as RTX Kernel Timer**. This selects the built-in SysTick timer with the processor clock as the clock source. In this case, the **RTOS Kernel Timer input clock frequency** should be **identical** to the CMSIS variable *SystemCoreClock* of the startup file *system_<device>.c*.

For details, refer to the online documentation section **Configuration of CMSIS-RTOS RTX – Tick Timer Configuration**.

CMSIS-RTOS User Code Templates

MDK provides user code templates you can use to create C source code for the application.

☞ In the **Project** window, right click a group, select **Add New Item to Group**, choose **User Code Template,** select any template and click **Add**.



CMSIS-RTOS RTX API Functions

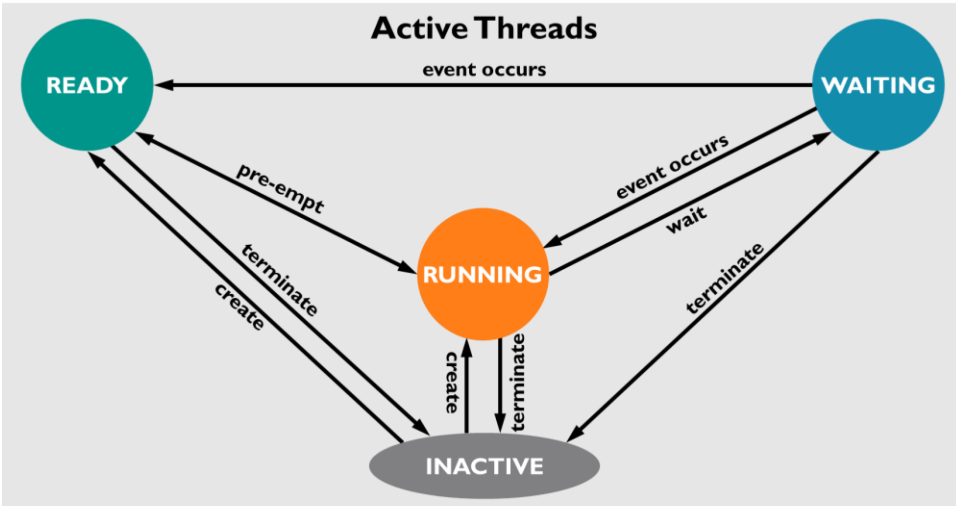
The table below lists the various API function categories that are available with the CMSIS-RTOS RTX.

API Category	Description
Thread Management	Define, create, and control thread functions.
Timer Management	Create and control timer and callback functions.
Signal Management	Control or wait for signal flags.
Mutex Management	Synchronize thread execution with a Mutex.
Semaphore Management	Control access to shared resources.
Memory Pool Management	Define and manage fixed-size memory pools
Message Queue Management	Control, send, receive, or wait for messages.
Mail Queue Management	Control, send, receive, or wait for mail.

TIP: The CMSIS-RTOS RTX tutorial available at **www.keil.com/pack/doc/CMSIS/RTX/html/index.html** explains the usage of the API functions.

Thread Management

The thread management functions allow you to define, create, and control your own thread functions in the system. The function *main()* is a special thread function that is started at system initialization and has the initial priority *osPriorityNormal*.



CMSIS-RTOS RTX assumes that threads are scheduled as shown in the figure above. Thread states change as described below:

A thread is created using the function *osThreadCreate()*. This puts the thread into the **READY** or **RUNNING** state (depending on the thread priority).

CMSIS-RTOS is pre-emptive. The active thread with the highest priority becomes the **RUNNING** thread provided it is not waiting for any event. The initial priority of a thread is defined with the *osThreadDef()* but may be changed during execution using the function *osThreadSetPriority()*.

The **RUNNING** thread transfers into the **WAITING** state when it is waiting for an event.

Active threads can be terminated any time using the function *osThreadTerminate()*. Threads can also terminate by exit from the usual *forever loop* and just a *return* from the thread function. Threads that are terminated are in the **INACTIVE** state and typically do not consume any dynamic memory resources.

Single Thread Program

A standard C program starts execution with the function *main()*. For an embedded application, this function is usually an endless loop and can be thought of as a single thread that is executed continuously.

Preemptive Thread Switching

Threads with the same priority need a round robin timeout or an explicit call of the *osDelay()* function to execute other threads. In the following example, if *job2* has a higher priority than *job1*, execution of *job2* starts instantly. *job2* preempts execution of *job1* (this is a very fast task switch requiring a few ms only).

Simple RTX Program using Round-Robin Task Switching

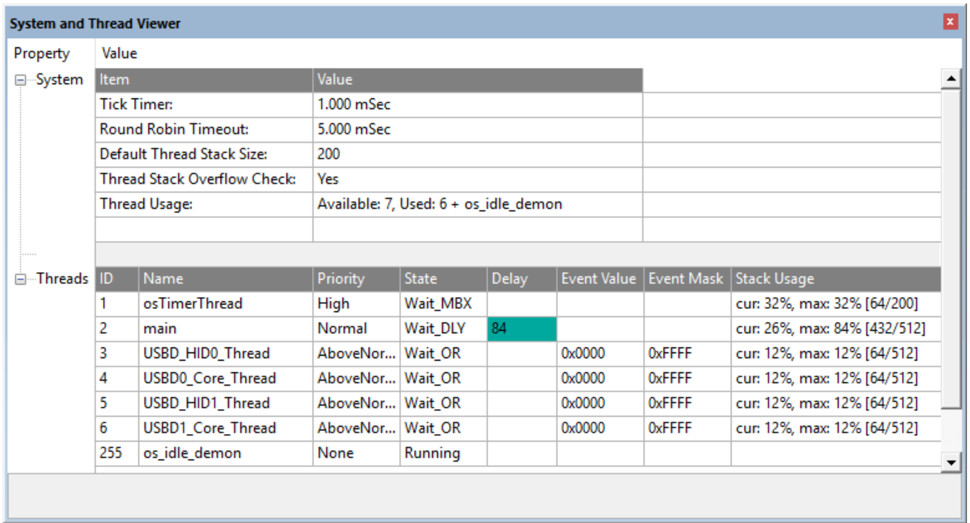
```
#include "cmsis_os.h"

int counter1;
int counter2;

void job1 (void const *arg) {
  while (1) {                                 // Loop forever
    counter1++;                               // Increment counter1
  }
}

void job2 (void const *arg) {
  while (1) {                                 // Loop forever
    counter2++;                               // Increment counter2
  }
}

osThreadDef (job1, osPriorityNormal, 1, 0);  // Define thread for job1
osThreadDef (job2, osPriorityNormal, 1, 0);  // Define thread for job2

int main (void) {                             // main() runs as thread
  osKernelInitialize ();                      // Initialize RTX

  osThreadCreate (osThread (job1), NULL);    // Create and start job1
  osThreadCreate (osThread (job2), NULL);    // Create and start job2

  osKernelStart ();                           // Start RTX kernel

  while (1) {
    osThreadYield ();                         // Next thread
  }
}
```

Start *job2* with Higher Thread Priority

```
 :
 osThreadDef (osThread (job2), osPriorityAboveNormal, 1, 0);
 :
```

CMSIS-RTOS System and Thread Viewer

The CMSIS-RTOS RTX Kernel has built-in support for RTOS aware debugging. During debugging, open **Debug → OS Support** and select **System and Thread Viewer**. This window shows system state information and the running threads.



System and Thread Viewer

Property	Value
System	

Item	Value
Tick Timer:	1.000 mSec
Round Robin Timeout:	5.000 mSec
Default Thread Stack Size:	200
Thread Stack Overflow Check:	Yes
Thread Usage:	Available: 7, Used: 6 + os_idle_demon

Threads

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Usage
1	osTimerThread	High	Wait_MBX				cur: 32%, max: 32% [64/200]
2	main	Normal	Wait_DLY	84			cur: 26%, max: 84% [432/512]
3	USBD_HID0_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
4	USBD0_Core_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
5	USBD_HID1_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
6	USBD1_Core_Thread	AboveNor...	Wait_OR		0x0000	0xFFFF	cur: 12%, max: 12% [64/512]
255	os_idle_demon	None	Running				

The **System** property shows general information about the RTOS configuration in the application. **Thread Usage** shows the number of available and threads and the used threads that are currently active.

The **Threads** property shows details about thread execution of the application. It shows for each thread information about priority, execution state and stack usage.

If the option **Stack usage watermark** is enabled for **Thread Configuration** in the file *RTX_Conf_CM.c*, the field **Stack Usage** shows **cur:** and **max:** stack load. The value **cur:** is the current stack usage at the actual program location. The value **max:** is the maximum stack load that occurred during thread execution, based on overwrites of the stack usage watermark pattern. This allows you to:

Identify stack overflows during thread execution or

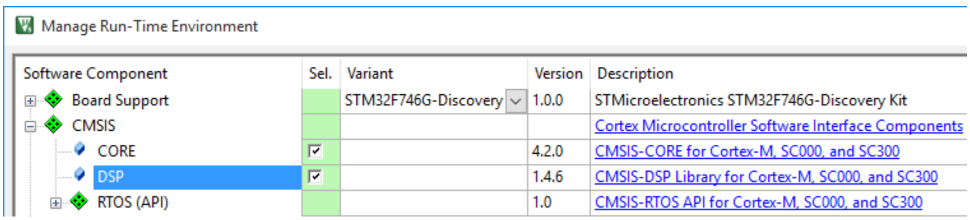
Optimize and reduce the stack space used for threads.

NOTE
*Using Trace, the debugger provides also a view with detailed timing information. Refer to **Event Viewer** on page 75 for more information.*

CMSIS-DSP

The CMSIS-DSP library is a suite of common digital signal processing (DSP) functions. The library is available in several variants optimized for different ARM Cortex-M processors.

When enabling the software component **::CMSIS:DSP** in the **Manage Run-Time Environment** dialog, the appropriate library for the selected device is automatically included into the project.


Manage Run-Time Environment

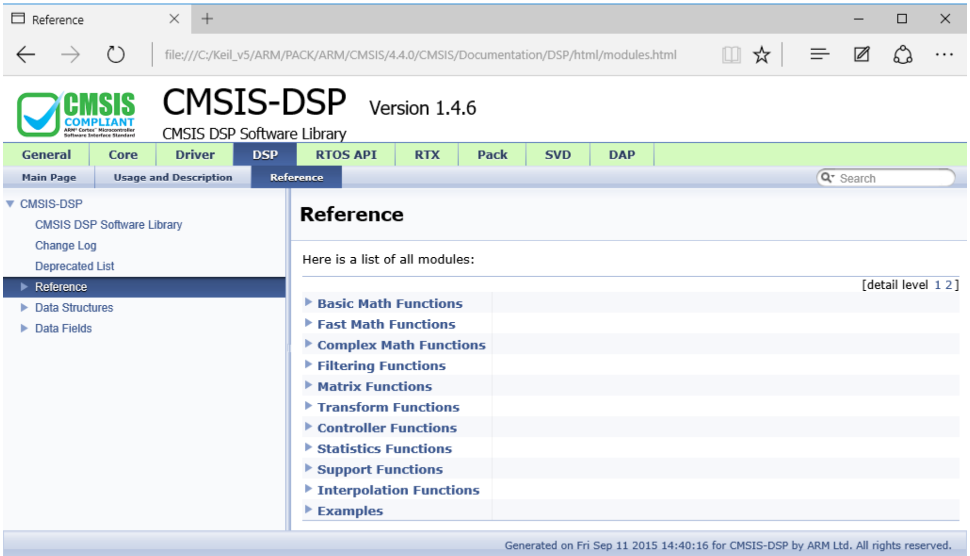
Software Component	Sel.	Variant	Version	Description
Board Support		STM32F746G-Discovery	1.0.0	STMicroelectronics STM32F746G-Discovery Kit
CMSIS				Cortex Microcontroller Software Interface Components
CORE	☑		4.2.0	CMSIS-CORE for Cortex-M, SC000, and SC300
DSP	☑		1.4.6	CMSIS-DSP Library for Cortex-M, SC000, and SC300
RTOS (API)			1.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300

The code example below shows the use of CMSIS-DSP library functions.

Multiplication of two matrixes using DSP functions

```
#include "arm_math.h"                    // ARM::CMSIS:DSP

const float32_t buf_A[9] = {             // Matrix A buffer and values
 1.0,  32.0,   4.0,
 1.0,  32.0,  64.0,
 1.0,  16.0,   4.0,
};

float32_t buf_AT[9];                     // Buffer for A Transpose (AT)
float32_t buf_ATmA[9] ;                  // Buffer for (AT * A)

arm_matrix_instance_f32 A;               // Matrix A
arm_matrix_instance_f32 AT;              // Matrix AT(A transpose)
arm_matrix_instance_f32 ATmA;            // Matrix ATmA( AT multiplied by A)

uint32_t rows = 3;                       // Matrix rows
uint32_t cols = 3;                       // Matrix columns

int main(void) {
  // Initialize all matrixes with rows, columns, and data array
  arm_mat_init_f32 (&A, rows, cols, (float32_t *)buf_A);  // Matrix A
  arm_mat_init_f32 (&AT, rows, cols, buf_AT);             // Matrix AT
  arm_mat_init_f32 (&ATmA, rows, cols, buf_ATmA);         // Matrix ATmA

  arm_mat_trans_f32 (&A, &AT);           // Calculate A Transpose (AT)
  arm_mat_mult_f32 (&AT, &A, &ATmA); // Multiply AT with A

  while (1);
}
```

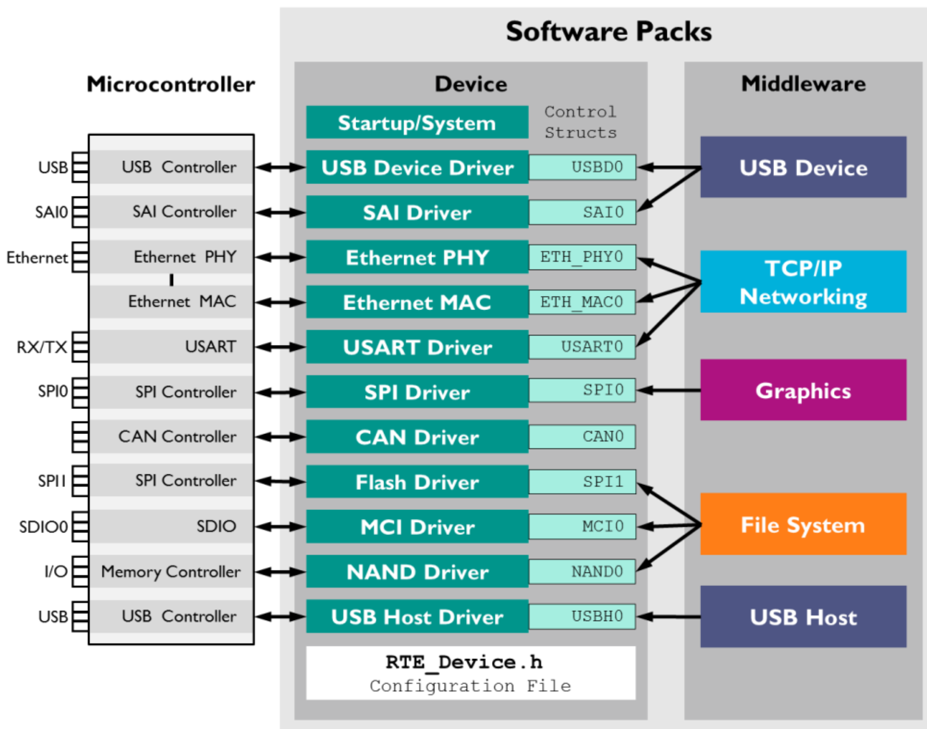
For more information, refer to the CMSIS-DSP documentation on www.keil.com/cmsis/dsp.



CMSIS-Driver

Device-specific **CMSIS-Drivers** provide the interface between the middleware and the microcontroller peripherals. These drivers are not limited to the MDK Middleware and are useful for various other middleware stacks to utilize those peripherals.

The device-specific drivers are usually part of the Software Pack that supports the microcontroller device and comply with the CMSIS-Driver standard. The Device Database on www.keil.com/dd2 lists drivers included in the Software Pack for the device.



Middleware components usually have various configuration files that connect to these drivers. For most devices, the *RTE_Device.h* file configures the drivers to the actual pin connection of the microcontroller device.

The middleware/application code connects to a driver instance via a *control struct*. The name of this *control struct* reflects the peripheral interface of the device. Drivers for most of the communication peripherals are part of the Software Packs that provide device support.

Use traditional C source code to implement missing drivers according the CMSIS-Driver standard.

Refer to www.keil.com/cmsis/driver for detailed information about the API interface of these CMSIS drivers.

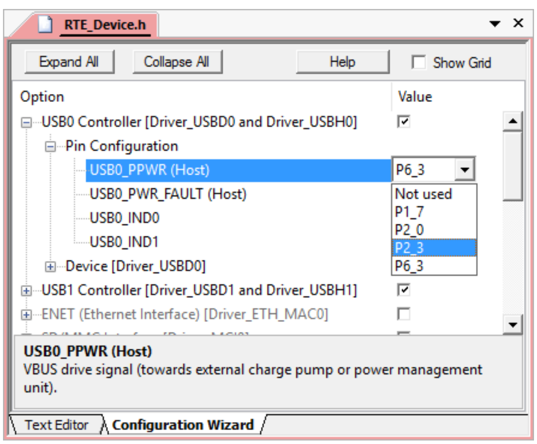
Configuration

There are multiple ways to configure a CMSIS-Driver. The classical method is using the *RTE_Device.h* file that comes with the device support.

Other devices may be configured using third party graphical configuration tools that allow the user to configure the device pin locations and the corresponding drivers. Usually, these configuration tools automatically create the required C code for import into the µVision project.

Using RTE_Device.h

For most devices, the *RTE_Device.h* file configures the drivers to the actual pin connection of the microcontroller device:



Using the Configuration Wizard view, you can configure the driver interfaces in a graphical mode without the need to edit manually the #defines in this header file.

Using STM32CubeMX

MDK supports CMSIS-Driver configuration using STM32CubeMX. This graphical software configuration tool allows you to generate C initialization code using graphical wizards for STMicroelectronics devices.

Simply select the required CMSIS-Driver in the Manage Run-Time Environment window and choose **Device:STM32Cube Framework (API):STM32CubeMX**. This will open STM32CubeMX for device and driver configuration. Once finished, generate the configuration code and import it into µVision.

For more information, visit the online documentation at **www.keil.com/pack/doc/STM32Cube/General/html/index.html**.

Validation

A Software Pack for CMSIS-Driver validation tests is available from **www.keil.com/pack**. It contains the source code and documentation of the CMSIS-Driver validation suite along with a required configuration file, and examples that shows the usage on various target platforms.

The CMSIS-Driver Validation Suite performs the following tests:

- Generic validation of API function calls
- Validation of configuration parameters
- Validation of communication with loopback tests
- Validation of communication parameters such as baudrate
- Validation of event functions

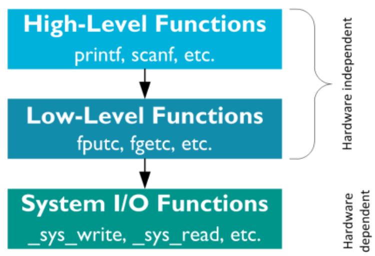
The test results can be printed to a console, output via ITM printf, or output to a memory buffer. Refer to the section **Driver Validation** in the CMSIS-Driver documentation available at **www.keil.com/cmsis/driver**.

Software Components

Compiler

The software component **Compiler** allows you to retarget I/O functions of the standard C run-time library. Application code frequently uses standard I/O library functions, such as *printf()*, *scanf()*, or *fgetc()* to perform input/output operations.

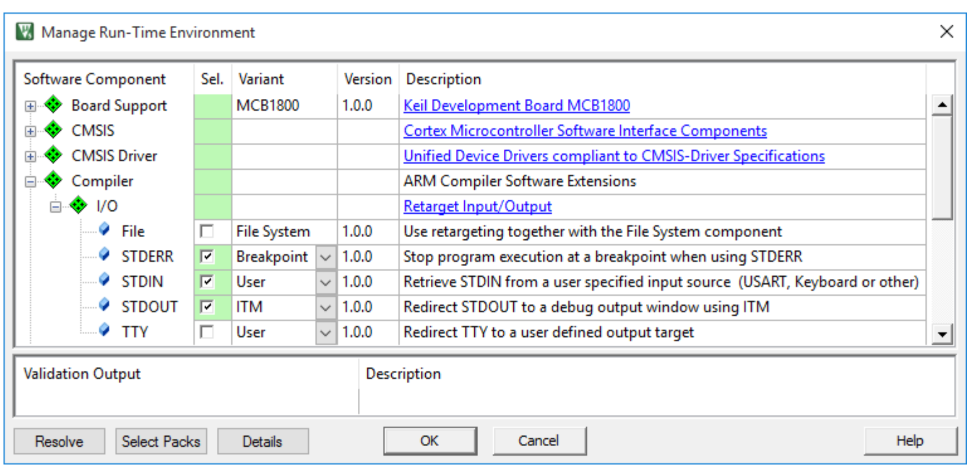
The structure of these functions in the standard ARM Compiler C run-time library is:



The high-level and low-level functions are not target-dependent and use the system I/O functions to interface with hardware.

The MicroLib of the ARM Compiler C run-time library interfaces with the hardware via low-level functions. The MicroLib implements a reduced set of high-level functions and therefore does not implement system I/O functions.

The software component **Compiler** retargets the I/O functions for the various standard I/O channels: File, STDERR, STDIN, STDOUT, and TTY:



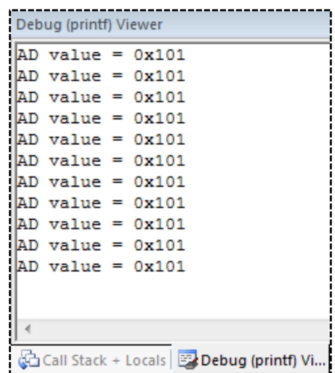
I/O Channel	Description
File	Channel for all file related operations (*fscanf, fprintf, fopen, fclose*, etc.)
STDERR	Standard error stream of the application to output diagnostic messages.
STDIN	Standard input stream going into the application (*scanf* etc.).
STDOUT	Standard output stream of the application (*printf* etc.).
TTY	Teletypewriter which is the last resort for error output.

The variant selection allows you to change the hardware interface of the I/O channel.

Variant	Description
File System	Use the File System component as the interface for File related operations
Breakpoint	When the I/O channel is used, the application stops with BKPT instruction.
ITM	Use Instrumentation Trace Macrocell (ITM) for I/O communication via the debugger.
User	Retarget I/O functions to a user defined routines (such as USART, keyboard).

The software component **Compiler** adds the file *retarget_io.c* that will be configured acording to the variant settings. For the **User** variant, user code templates are available that help you to implement your own functionality. Refer to the documentation for more information.

ITM in the Cortex-M3/M4/M7 supports *printf* style debugging. If you choose the variant **ITM**, the I/O library functions perform I/O operations via the **Debug (printf) Viewer** window.



Board Support

There are a couple of interfaces that are frequently used on development boards, such as LEDs, push buttons, joysticks, A/D and D/A converters, LCDs, and touchscreens as well as external sensors such as thermometers, accelerometers, magnetometers, and gyroscopes.

The **Board Support Interface API** provides standardized access to these interfaces. This enables software developers to concentrate on their application code instead of checking device manuals for register settings to toggle a particular GPIO.

Many Device Family Packs (DFPs) have board support included. You can choose board support from the Manage Run-Time Environment window:

Software Component	Sel.	Variant	Version	Description
Board Support		STM32F746G-Discovery	1.0.0	STMicroelectronics STM32F746G-Discovery Kit
Buttons (API)			1.00	Buttons Interface
Buttons	☑		1.0.0	Buttons Interface for STMicroelectronics STM32F746G-Discovery Kit
Drivers				Kinetis BSP Drivers
Graphic LCD (API)			1.00	Graphic LCD Interface
LED (API)			1.00	LED Interface
LED	☑		1.0.0	LED Interface for STMicroelectronics STM32F746G-Discovery Kit
Touchscreen (API)			1.00	Touchscreen Interface
emWin LCD (API)			1.1	emWin LCD Interface

Be sure to select the correct **Variant** to enable the correct pin configurations for your particular development board.

You can add board support to your custom board by creating the required support files for your board’s Software Pack. Refer to the API documentation available at: **http://www.keil.com/pack/doc/mw/Board/html/index.html**

Create Applications

This chapter guides you through the steps required to create and modify projects using CMSIS described in the previous chapter.

NOTE
The example code in this section works for the MCB1800 evaluation board (populated with LPC1857). Adapt the code for other starter kits or boards.

The tutorial creates the project *Blinky* in the two basic design concepts:

RTOS design using CMSIS-RTOS RTX.

Infinite loop design for bare-metal systems without RTOS Kernel.

Blinky with CMSIS-RTOS RTX

The section explains the creation of the project using the following steps:

Setup the Project: create a project file and select the microcontroller device along with the relevant CMSIS components.

Configure the Device Clock Frequency: configure the system clock.

Customize the CMSIS-RTOS RTX Kernel: adapt the RTOS kernel.

Create the Source Code Files: add and create the application files.

Build the Application Image: compile and link the application for downloading it to an on-chip Flash memory of a microcontroller device.

Using the Debugger on page 64 guides you through the steps to connect your evaluation board to the PC and to download the application to the target.

For the project *Blinky*, you will create the following application files:

main.c This file contains the *main()* function that initializes the RTOS kernel, the peripherals, and starts thread execution.

LED.c The file contains functions to initialize and control the GPIO port and the thread function *blink_LED()*. The *LED_Initialize()* function initializes the GPIO port pin. The functions *LED_On()* and *LED_Off()* control the port pin that interfaces to the LED.

LED.h The header file contains the function prototypes for the functions in *LED.c* and is included into the file *main.c*.

Setup the Project

From the µVision menu bar, choose **Project – New µVision Project**.

Select an empty folder and enter the project name, for example, *Blinky*. Click **Save**, which creates an empty project file with the specified name (*Blinky.uvproj*).

Next, the dialog **Select Device for Target** opens.

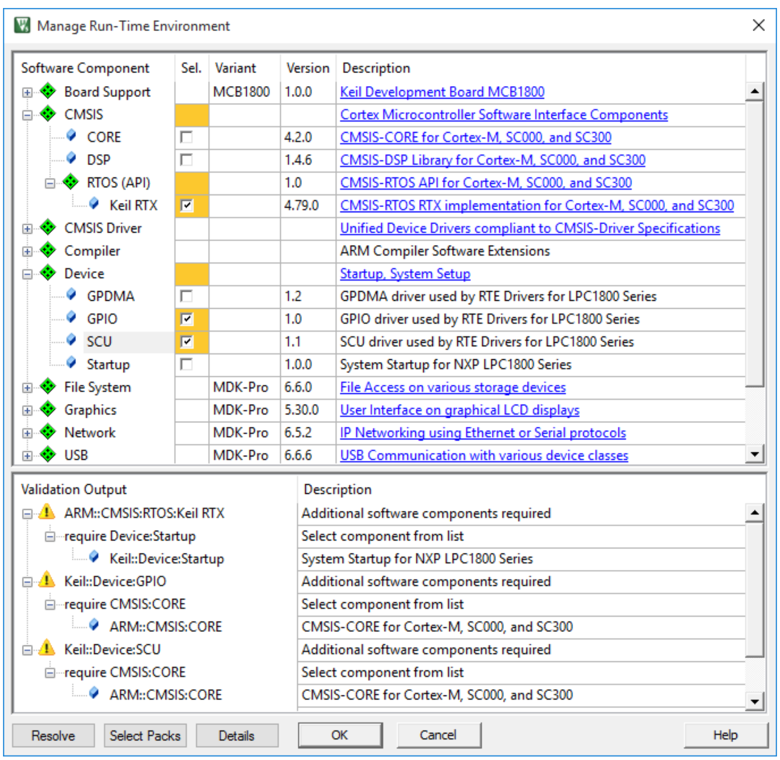
Select the LPC1857 and click **OK**.

The device selection defines essential tool settings such as compiler controls, the memory layout for the linker, and the Flash programming algorithms.

The **Manage Run-Time Environment** dialog opens and shows the software components that are installed and available for the selected device.

Expand **::CMSIS:RTOS(API)** and enable **:Keil RTX**.

Expand **::Device** and enable **:GPIO** and **:SCU**.



The **Validation Output** field shows dependencies to other software components. In this case, the component **::Device:Startup** is required.

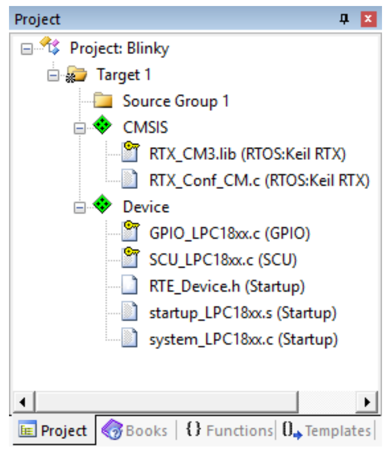
TIP: A click on a message highlights the related software component.

☞ Click **Resolve**.

This resolves all dependencies and enables other required software components (here, **::CMSIS:Core** and **::Device:Startup**).

☞ Click **OK**.

The selected software components are included into the project together with the startup file, the RTX configuration file, and the CMSIS system files. The **Project** window displays the selected software components along with the related files. Double-click on a file to open it in the editor.



Configure the Device Clock Frequency

The system or core clock is defined in the *system_<device>.c* file. The core clock is also the input clock for the RTOS Kernel Timer and, therefore, the RTX configuration file needs to match this setting.

NOTE
Some devices perform the system setup as part of the main function and/or use a software framework that is configured with external utilities.

Refer to ***Device Startup Variations*** *on page 56 for more information.*

The clock configuration for an application depends on various factors such as the clock source (XTAL or on-chip oscillator), and the requirements for memory and peripherals. Silicon vendors provide the device-specific file *system_<device>.c* and therefore it is required to read the related documentation.

TIP: Open the reference manual from the **Books** window for detailed information about the microcontroller clock system.

The MCB1800 development kit runs with an external 12 MHz XTAL. The PLL generates a core clock frequency of 180 MHz. As this is the default, no modifications are necessary. However, you can change the settings for your custom development board in the file *system_LPC18xx.c*.

☞ To edit the file *system_LPC18xx.c*, expand the group **Device** in the **Project** window, double-click on the file name, and modify the code as shown below.

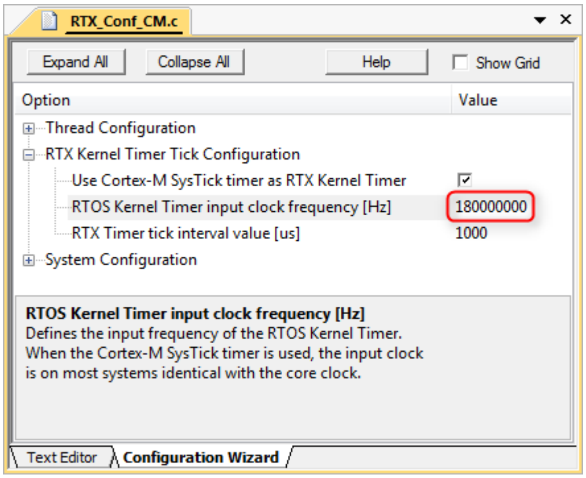
Set PLL Parameters in *system_LPC18xx.c*

```
:
/* PLL1 output clock: 180MHz, Fcco: 180MHz, N = 1, M = 15, P = x        */
#define PLL1_NSEL   0           /* Range [0 -   3]: Pre-divider  ratio N */
#define PLL1_MSEL  14           /* Range [0 - 255]: Feedback-div ratio M */
#define PLL1_PSEL   0           /* Range [0 -   3]: Post-divider ratio P */

#define PLL1_BYPASS 0           /* 0: Use PLL, 1: PLL is bypassed        */
#define PLL1_DIRECT 1           /* 0: Use PSEL, 1: Don't use PSEL        */
#define PLL1_FBSEL  0           /* 0: FCCO is used as PLL feedback       */
                                /* 1: FCLKOUT is used as PLL feedback    */
:
```

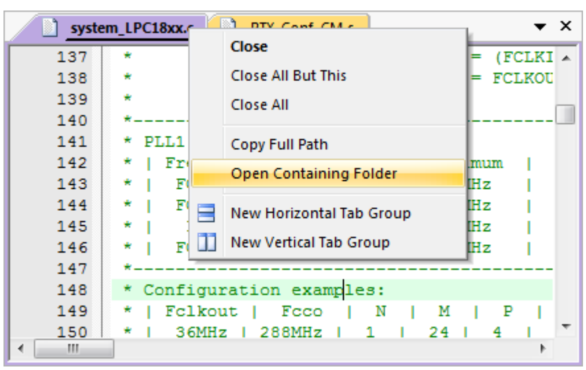
Customize the CMSIS-RTOS RTX Kernel

☞ In the **Project** window, expand the group **CMSIS**, open the file *RTX_Conf_CM.c*, and click the tab **Configuration Wizard** at the bottom of the editor.

Expand **RTX Kernel Timer Tick Configuration** and set the **Timer clock value** to match the core clock.



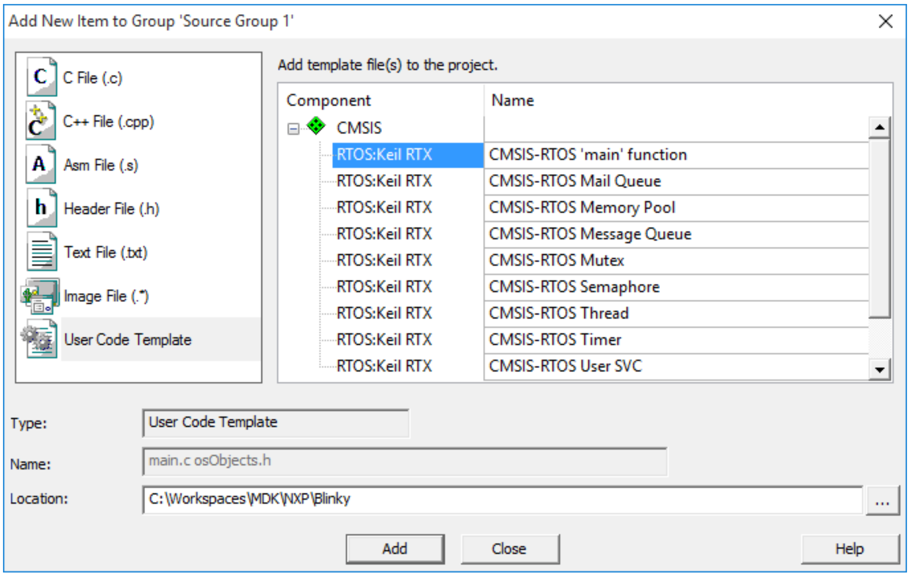
TIP: You may copy the compiler define settings and *system_<device>.c* from example projects. Right click on the filename in the editor and use **Open Containing Folder** to locate the file.



Create the Source Code Files

Add your application code using pre-configured **User Code Templates** containing routines that resemble the functionality of the software component.

☞ In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**.



☞ Click on **User Code Template** to list available code templates for the software components included in the project. Select **CMSIS-RTOS 'main' function** and click **Add**.

This adds the file *main.c* to the project group **Source Group 1**. Now you can add application specific code to this file.

☞ Right-click on a blank line in the file *main.c* and select **Insert '#include files'**. Include the header file *LPC18xx.h* for the selected device.

Then, add the code below to create a function *blink_LED()* that blinks LEDs on the evaluation kit. Define *blink_LED()* as an RTOS thread using *osThreadDef()* and start it with os*ThreadCreate()*.

Code for *main.c*

```
/*----------------------------------------------------------------------------
 * CMSIS-RTOS 'main' function template
 *---------------------------------------------------------------------------*/

#define osObjectsPublic           // Define objects in main module
#include "osObjects.h"            // RTOS object definitions
#include "LPC18xx.h"              // Device header
#include "LED.h"                  // Initialize and set GPIO Port

/*
 * main: initialize and start the system
 */
int main (void) {
  osKernelInitialize ();          // Initialize CMSIS-RTOS
  // initialize peripherals here
  LED_Initialize ();              // Initialize LEDs

  // create 'thread' functions that start executing,
  // example: tid_name = osThreadCreate (osThread(name), NULL);
  Init_BlinkyThread ();           // Start Blinky thread
  osKernelStart ();               // Start thread execution

  while (1);
}
```

☞ Create an empty C-file named *LED.c* using the dialog **Add New Item to Group** and add the code to initialize and access the GPIO port pins that control the LEDs.

Code for *LED.c*

```
/*----------------------------------------------------------------------------
 * File LED.c
 *---------------------------------------------------------------------------*/
#include "SCU_LPC18xx.h"
#include "GPIO_LPC18xx.h"
#include "cmsis_os.h"                   // ARM::CMSIS:RTOS:Keil RTX

void blink_LED (void const *argument);  // Prototype function

osThreadDef (blink_LED, osPriorityNormal, 1, 0); // Define blinky thread

void LED_Initialize (void) {
  GPIO_PortClock     (1);               // Enable GPIO clock

  /* Configure pin: Output Mode with Pull-down resistors */
  SCU_PinConfigure (13, 10, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir       (6, 24, GPIO_DIR_OUTPUT);
  GPIO_PinWrite     (6, 24, 0);
}

void LED_On (void) {
  GPIO_PinWrite      (6, 24, 1);         // LED on: set port
}

void LED_Off (void) {
  GPIO_PinWrite      (6, 24, 0);         // LED off: clear port
}

// Blink LED function
void blink_LED(void const *argument) {
  for (;;) {
    LED_On ();                          // Switch LED on
    osDelay (500);                      // Delay 500 ms
    LED_Off ();                         // Switch off
    osDelay (500);                      // Delay 500 ms
  }
}

void Init_BlinkyThread (void) {
  osThreadCreate (osThread(blink_LED), NULL);    // Create thread
}
```

NOTE

You can also use the functions as provided by the ***Board Support*** *component described on* *page 44.*

Create an empty header file named *LED.h* using the dialog **Add New Item to Group** and define the function prototypes of *LED.c*.

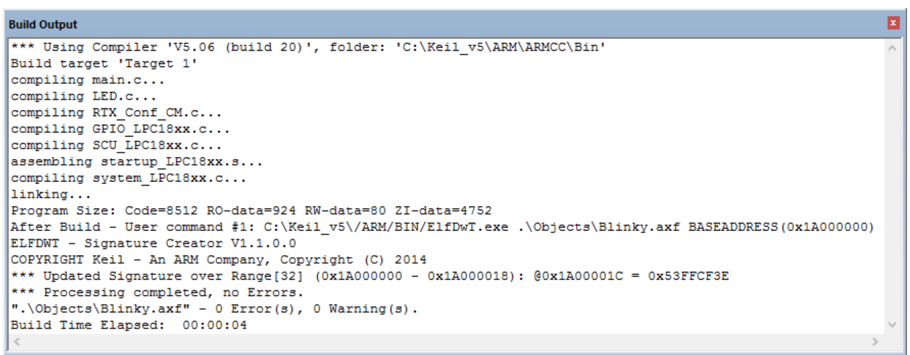
Code for *LED.h*

```
/*----------------------------------------------------------------------
 * File LED.h
 *---------------------------------------------------------------------*/
void LED_Initialize ( void );                 // Initialize GPIO
void LED_On ( void );                         // Switch Pin on
void LED_Off ( void );                        // Switch Pin off

void blink_LED ( void const *argument );      // Blink LEDs in a thread
void Init_BlinkyThread ( void );              // Initialize thread
```

Build the Application Image

Build the application, which compiles and links all related source files.

Build Output shows information about the build process. An error-free build displays program size information, zero errors, and zero warnings.



Build Output

```
*** Using Compiler 'V5.06 (build 20)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Target 1'
compiling main.c...
compiling LED.c...
compiling RTX_Conf_CM.c...
compiling GPIO_LPC18xx.c...
compiling SCU_LPC18xx.c...
assembling startup_LPC18xx.s...
compiling system_LPC18xx.c...
linking...
Program Size: Code=8512 RO-data=924 RW-data=80 ZI-data=4752
After Build - User command #1: C:\Keil_v5\/ARM/BIN/ElfDwT.exe .\Objects\Blinky.axf BASEADDRESS(0x1A000000)
ELFDWT - Signature Creator V1.1.0.0
COPYRIGHT Keil - An ARM Company, Copyright (C) 2014
*** Updated Signature over Range[32] (0x1A000000 - 0x1A000018): @0x1A00001C = 0x53FFCF3E
*** Processing completed, no Errors.
".\Objects\Blinky.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:04
```

The section **Using the Debugger** on page 64 guides you through the steps to connect your evaluation board to the workstation and to download the application to the target hardware.

TIP: You can verify the correct clock and RTOS configuration settings of the target hardware by checking the one-second interval of the LED.

Blinky with Infinite Loop Design

Based on the previous example, we create a Blinky application with the infinite loop design and without using CMSIS-RTOS RTX functions. The project contains the user code files:

main.c This file contains the *main()* function, the function *Systick_Init()* to initialize the System Tick Timer and its handler function *SysTick_Handler()*. The function *Delay()* waits for a certain time.

LED.c The file contains functions to initialize the GPIO port pin and to set the port pin on or off. The function *LED_Initialize()* initializes the GPIO port pin. The functions *LED_On()* and *LED_Off()* enable or disable the port pin.

LED.h The header file contains the function prototypes created in *LED.c* and must be included into the file *main.c*.

Open the **Manage Run-Time Environment** and deselect the software component **::CMSIS:RTOS (API):Keil RTX**.

☞ Open the file *main.c* and add the code to initialize the System Tick Timer, write the System Tick Timer Interrupt Handler, and the delay function.

```
/*----------------------------------------------------------------------------
 * file main.c
 *---------------------------------------------------------------------------*/

#include "LPC18xx.h"                  // Device header
#include "LED.h"                      // Initialize and set GPIO Port

int32_t volatile msTicks = 0;         // Interval counter in ms

// Set the SysTick interrupt interval to 1ms
void SysTick_Init (void) {
  if (SysTick_Config (SystemCoreClock / 1000)) {
    // handle error
  }
}

// SysTick Interrupt Handler function called automatically
void SysTick_Handler (void) {
  msTicks++;                          // Increment counter
}

// Wait until msTick reaches 0
void Delay (void) {
  while (msTicks < 499);              // Wait 500ms
  msTicks = 0;                        // Reset counter
}
```

```
int main (void) {
  // initialize peripherals here
  LED_Initialize ();                    // Initialize LEDs
  SystemCoreClockUpdate();              // Update SystemCoreClock to 180 MHz
  SysTick_Init ();                      // Initialize SysTick Timer

  while (1) {
    LED_On ();                          // Switch on
    Delay ();                           // Delay
    LED_Off ();                         // Switch off
    Delay ();                           // Delay
  }
}
```

☞ Open the file *LED.c* and remove unnecessary functions. The code should look like this.

```
/*-------------------------------------------------------------------------
 * File LED.c
 *-----------------------------------------------------------------------*/
#include "SCU_LPC18xx.h"
#include "GPIO_LPC18xx.h"

void LED_Initialize (void) {

GPIO_PortClock     (1);                        // Enable GPIO clock

/* Configure pin: Output Mode with Pull-down resistors */
SCU_PinConfigure (13, 10, (SCU_CFG_MODE_FUNC4 | SCU_PIN_CFG_PULLDOWN_EN));
GPIO_SetDir      (6, 24, GPIO_DIR_OUTPUT);
GPIO_PinWrite    (6, 24, 0);
}

void LED_On (void) {
GPIO_PinWrite     (6, 24, 1);                  // LED on: set port
}

void LED_Off (void) {
GPIO_PinWrite     (6, 24, 0);                  // LED off: clear port
}
```

☞ Open the file *LED.h* and modify the code.

```
/*-------------------------------------------------------------------------
 *  file: LED.h
 *-----------------------------------------------------------------------*/
void LED_Initialize (void);                    // Initialize LED Port Pins
void LED_On (void);                            // Set LED on
void LED_Off (void);                           // Set LED off
```

Build the Application Image

 Build the application, which compiles and links all related source files.

The section **Using the Debugger** on page 64 guides you through the steps to connect your evaluation board to the PC and to download the application to the target hardware.

TIP: You can verify the correct clock configuration of the target hardware by checking the one-second interval of the LED.

Device Startup Variations

Some devices perform a significant part of the system setup as part of the device hardware abstraction layer (HAL) and therefore the device initialization is done from within the main function. Such devices frequently use a software framework that is configured with external utilities.

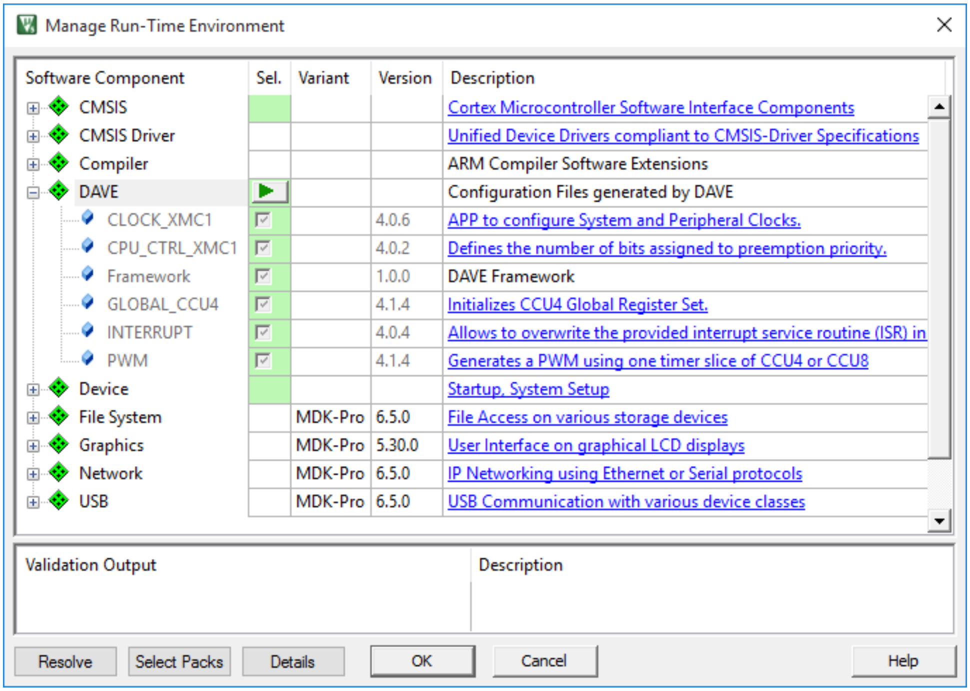
The **::Device** software component may contain therefore additional components that are required to startup the device. Refer to the online help system for further information. In the following section, device startup variations are exemplified.

Example: Infineon XMC1000 using DAVE

Using Infineon's **DAVE™**, you can automatically generate code based on so-called DAVE Apps. Within the Eclipse-based IDE, you can add, configure, and connect the apps to suit your application. During this process, you will configure the clock settings using the **CLOCK_XMC_1_0** app (in case of the XMC1000 family). This app sets the correct registers within the core to reach the desired frequency. At the end of the generated code, it calls the CMSIS function *SystemCoreClockUpdate()*.

All steps to import a DAVE project into µVision are explained in the application note 258 available at **http://www.keil.com/appnotes/docs/apnt_258.asp**.

After µVision imported the project, the **Manage Run-Time Environment** window shows the group **::DAVE3** with the generated apps as components.



Inside µVision, the component **::DAVE** is locked. Use the s**tart button** to open DAVE for changing the configuration of the apps.

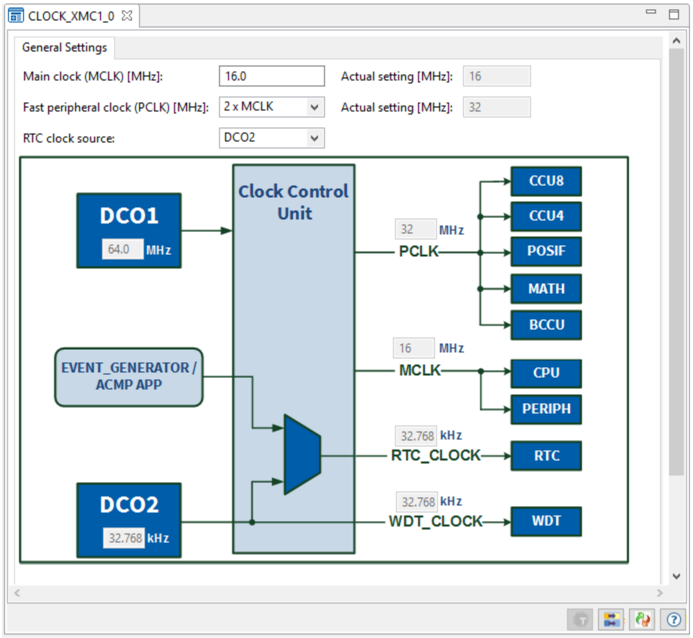
The *clock_xmc1_conf.c* file contains a data structure for setting the clock registers. The following is an example that shows how DAVE sets the values according to the configuration from within the tool:

Code for clock_xmc1_conf.c

```
:
/**********************************************************************
* DATA STRUCTURES
**********************************************************************/
const XMC_SCU_CLOCK_CONFIG_t CLOCK_XMC1_0_CONFIG =
{
.pclk_src = XMC_SCU_CLOCK_PCLKSRC_DOUBLE_MCLK,
.rtc_src = XMC_SCU_CLOCK_RTCCLKSRC_DCO2,
.fdiv = 0U,  /**< Fractional divider */
.idiv = 1U,  /**< 8Bit integer divider */
};
```

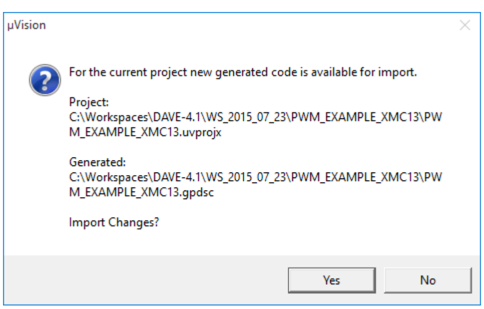
Change the Clock Setup using DAVE

If you need to change these clock values, open the **Manage Run-Time Environment** window and press the **start button** to open DAVE. Use **Configure APP Instance…** to change the clock settings:



Re-run the code generation in DAVE.

This will change the generated files, which will be recognized by µVision automatically:



Click on **Yes** to reload the changed file.

Example: STM32Cube

Many STM32 devices are using the **STM32Cube Framework** that can be configured with a classical method using the *RTE_Device.h* configuration file or by using **STM32CubeMX**.

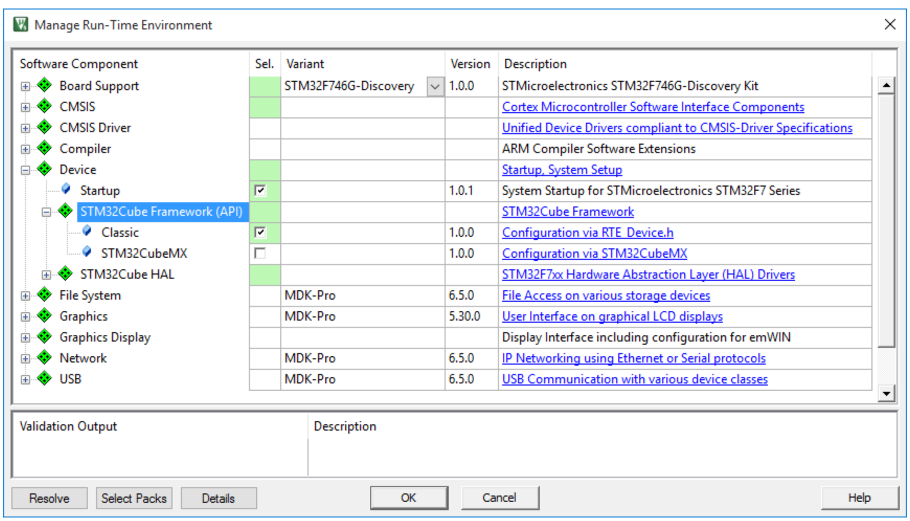
The classic **STM32Cube Framework** component provides a specific user code template that implements the system setup. Using **STM32CubeMX**, the *main.c* file and other source files required for startup are copied into the project below the **STM32CubeMX:Common Sources** group.

Setup the Project using the Classic Framework

This example creates a project for the STM32F746G-Discovery kit using the classical method. In the **Manage Run-Time Environment** window, select the following:

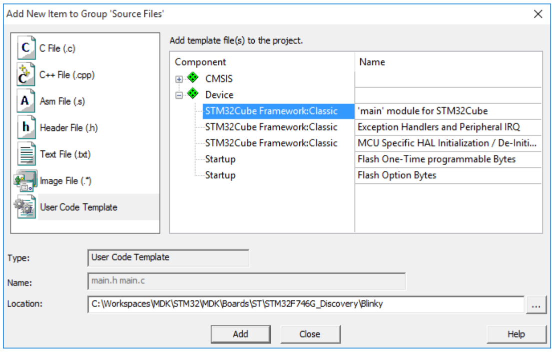
☞ Expand **::Device:STM32Cube Framework (API)** and enable **:Classic**.

Expand **::Device** and enable **:Startup**.



☞ Click **Resolve** to enable other required software components and then **OK**.

In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**.



Click on **User Code Template** to list available code templates for the software components included in the project. Select **'main' module for STM32Cube** and click **Add**.

The *main.c* file contains the function *SystemClock_Config()*. Here, you need to make the settings for the clock setup:

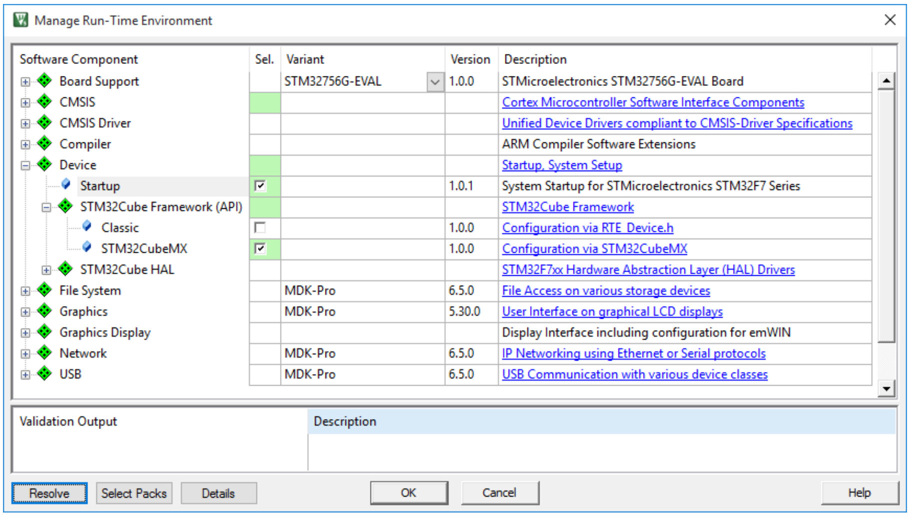
Code for *main.c*

```
:
static void SystemClock_Config (void) {
RCC_ClkInitTypeDef RCC_ClkInitStruct;
RCC_OscInitTypeDef RCC_OscInitStruct;

/* Enable HSE Oscillator and activate PLL with HSE as source */
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
RCC_OscInitStruct.HSEState = RCC_HSE_ON;
RCC_OscInitStruct.HSIState = RCC_HSI_OFF;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
RCC_OscInitStruct.PLL.PLLM = 25;
RCC_OscInitStruct.PLL.PLLN = 432;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 9;
HAL_RCC_OscConfig(&RCC_OscInitStruct);

/* Activate the OverDrive to reach the 216 MHz Frequency */
HAL_PWREx_EnableOverDrive();

/* Select PLL as system clock source and configure the HCLK, PCLK1 and
PCLK2 clocks dividers */
RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK |
RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_7);
}
:
```

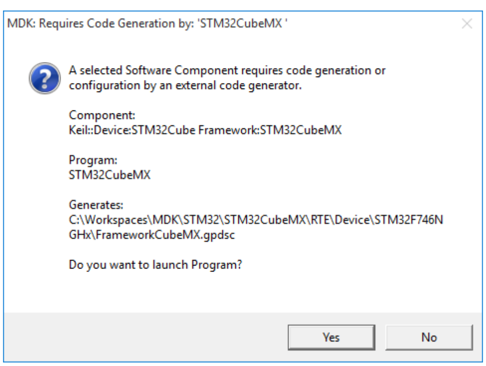
Setup the Project using STM32CubeMX

This example creates the same project as before using **STM32CubeMX**. In the **Manage Run-Time Environment** window, select the following:

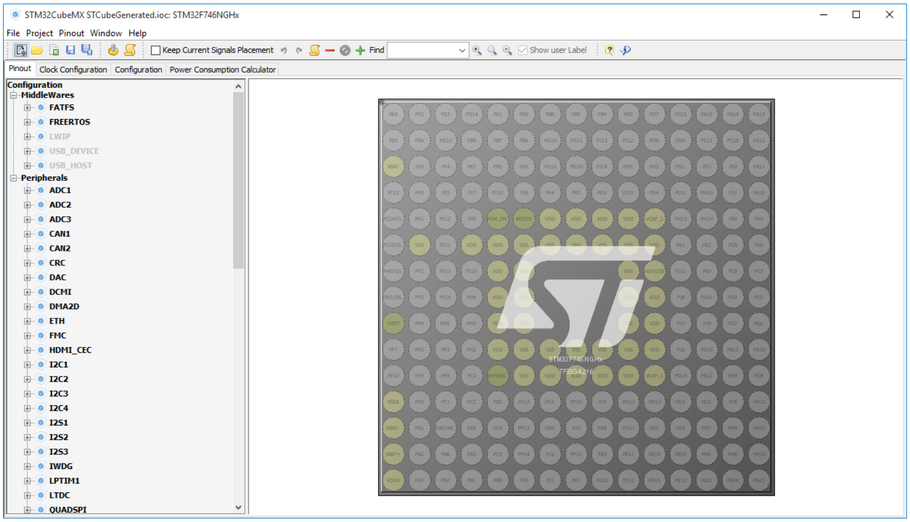
☞ Expand **::Device:STM32Cube Framework (API)** and enable **:STM32CubeMX**. Expand **::Device** and enable **:Startup**.



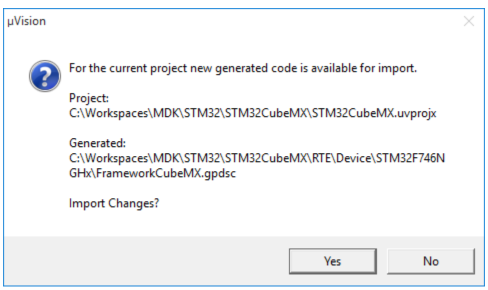
☞ Click **Resolve** to enable other required software components and then **OK**. A new window will ask you to start **STM32CubeMX**.



STM32CubeMX is started with the correct device selected:



☞ Configure your device as required. When done, go to **Project → Generate Code** to create a GPDSC file. µVision will notify you:



☞ Click Yes to import the project. The **main.c** and other generated files are added to a folder called **STM32CubeMX:Common Sources**.

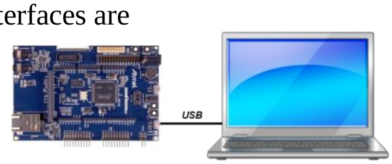
Debug Applications

The ARM CoreSight™ technology integrated into the ARM Cortex-M processor based devices provides powerful debug and trace capabilities. It enables run-control to start and stop programs, breakpoints, memory access, and Flash programming. Features like sampling, data trace, exceptions including program counter (PC) interrupts, and instrumentation trace are available in most devices. Devices integrate instruction trace using ETM, ETB, or MTB to enable analysis of the program execution. Refer to **www.keil.com/coresight** for a complete overview of the debug and trace capabilities.

Debugger Connection

MDK contains the µVision Debugger that connects to various debug/trace adapters, and allows you to program the Flash memory. It supports traditional features like simple and complex breakpoints, watch windows, and execution control. Using trace, additional features like event/exception viewers, logic analyzer, execution profiler, and code coverage are supported.

The ULINK2 and ULINK-ME debug adapters interface to JTAG/SWD debug connectors and support trace with the Serial Wire Output (SWO). The ULINK*pro* debug/trace adapter also interfaces to ETM trace connectors and uses streaming trace technology to capture the complete instruction trace for code coverage and execution profiling. Refer to **www.keil.com/ulink** for more information.



CMSIS-DAP based USB JTAG/SWD debug interfaces are typically part of an evaluation board or starter kit and offer integrated debug features. MDK also supports several proprietary interfaces that offer a similar technology.

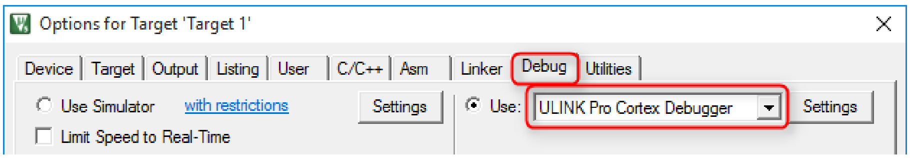
MDK connects to third-party debug solutions such as Segger J-Link or J-Trace. Some starter kit boards provide the J-Link Lite technology as an on-board solution.

Using the Debugger

Next, you will debug the *Blinky* application created in the previous chapter on hardware. You need to configure the debug connection and Flash programming utility.

Select the debug adapter and configure debug options.

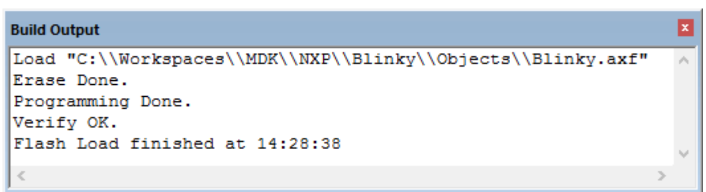
From the toolbar, choose **Options for Target**, click the **Debug** tab, enable **Use**, and select the applicable debug driver.



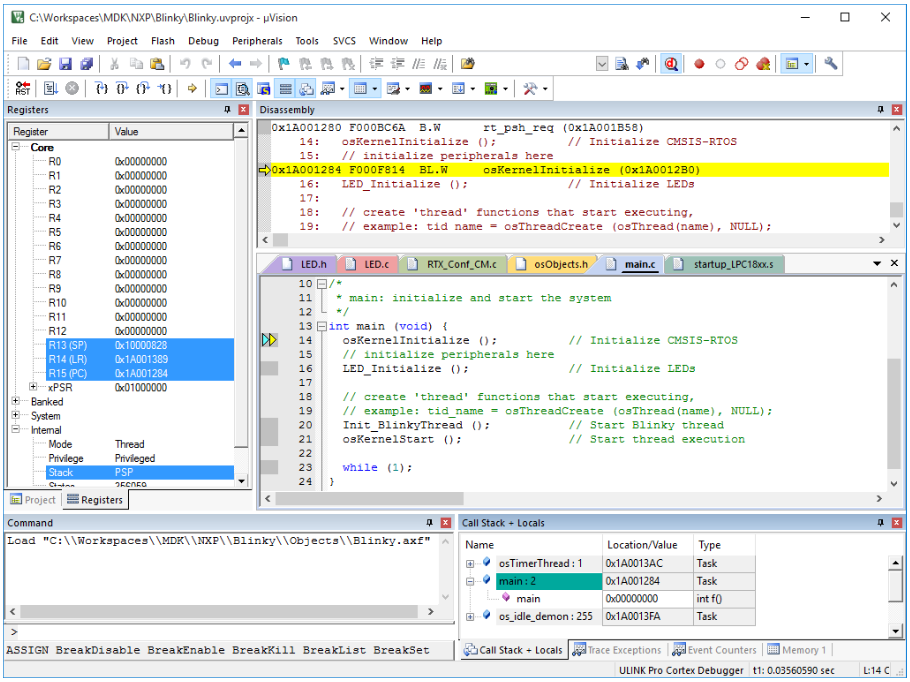
The device selection already configures the Flash programming algorithm for on-chip memory. Verify the configuration using the **Settings** button.

Program the application into Flash memory.

From the toolbar, choose **Download.** The **Build Output** window shows messages about the download progress.



Start debugging on hardware. From the toolbar, select **Start/Stop Debug Session**.



During the start of a debugging session, µVision loads the application, executes the startup code, and stops at the main C function.

Click **Run** on the toolbar. The LED flashes with a frequency of one second.

Debug Toolbar

The debug toolbar provides quick access to many debugging commands such as:

Step steps through the program and into function calls.

Step Over steps through the program and over function calls.

Step Out steps out of the current function.

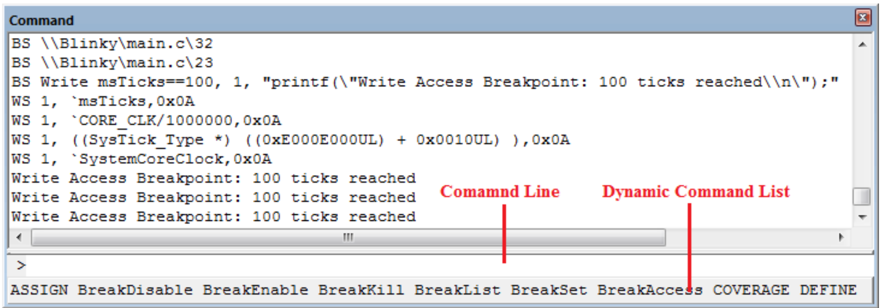
Stop halts program execution.

Reset performs a CPU reset.

Show to the statement that executes next (current PC location).

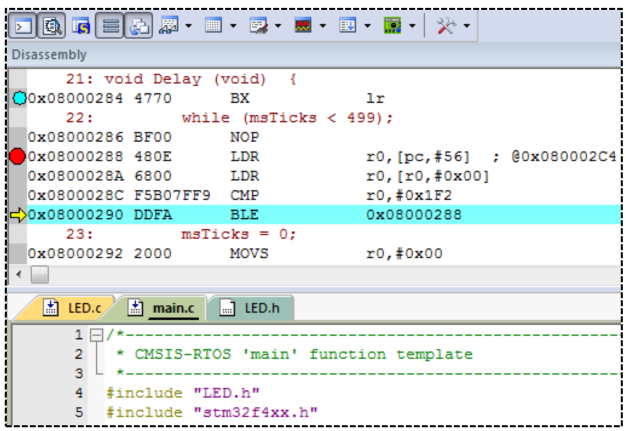
Command Window

You may also enter debug commands in the **Command** window.



On the **Command Line** enter debug commands or press **F1** to access detailed help information.

Disassembly Window



The **Disassembly** window shows the program execution in assembly code intermixed with the source code (when available). When this is the active window, then all debug stepping commands work at the assembly level.

The window margin shows markers for breakpoints, bookmarks, and for the next execution statement.

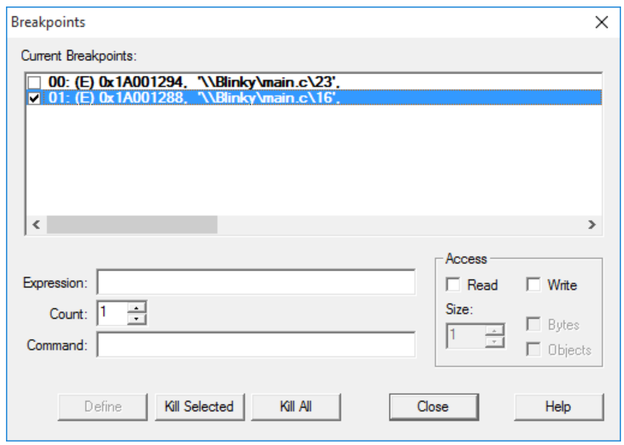
Breakpoints

You can set breakpoints

- While creating or editing your program source code. Click in the grey margin of the editor or **Disassembly** window to set a breakpoint.
- Using the breakpoint buttons in the toolbar.
- Using the menu Debug – Breakpoints.
- Entering commands in the **Command** window.
- Using the context menu of the **Disassembly** window or editor.

Breakpoints Window

You can define sophisticated breakpoints using the **Breakpoints** window.



Open the **Breakpoints** window from the menu **Debug**.

Enable or disable breakpoints using the checkbox in the field **Current Breakpoints**. Double-click on an existing breakpoint to modify the definition.

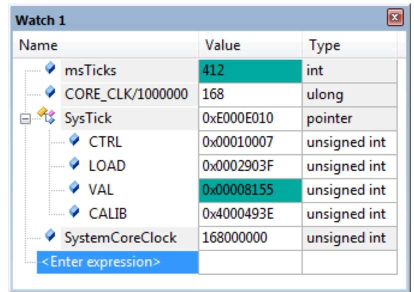
Enter an **Expression** to add a new breakpoint. Depending on the expression, one of the following breakpoint types is defined:

- **Execution Breakpoint (E)**: is created when the expression specifies a code address and triggers when the code address is reached.
- **Access Breakpoint (A)**: is created when the expression specifies a memory access (read, write, or both) and triggers on the access to this memory address. Use a compare (==) operator to compare for a specified value.

If a **Command** is specified for a breakpoint, µVision executes the command and resumes executing the target program.

The **Count** value specifies the number of times the breakpoint expression is true before the breakpoint halts program execution.

Watch Window



The **Watch** window allows you to observe program symbols, registers, memory areas, and expressions.

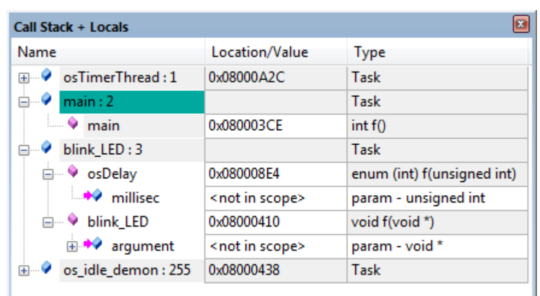
Open a **Watch** window from the toolbar or the menu using **View – Watch Windows**.

Add variables to the **Watch** window with:

- Click on the field **<Enter expression>** and double-click or press **F2**.
- In the Editor when the cursor is located on a variable, use the context menu select **Add <*item name*> to…**
- Drag and drop a variable into a **Watch** window.
- In the **Command** window, use the **WATCHSET** command.

The window content is updated when program execution is halted, or during program execution when **View – Periodic Window Update** is enabled.

Call Stack and Locals Window

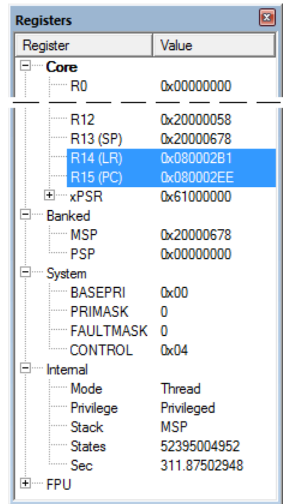


The **Call Stack + Locals** window shows the function nesting and variables of the current program location.

Open the **Call Stack + Locals** window from the toolbar or the menu using **View – Call Stack Window**.

When program execution stops, the **Call Stack + Locals** window automatically shows the current function nesting along with local variables. Threads are shown for applications that use the CMSIS-RTOS RTX.

Register Window



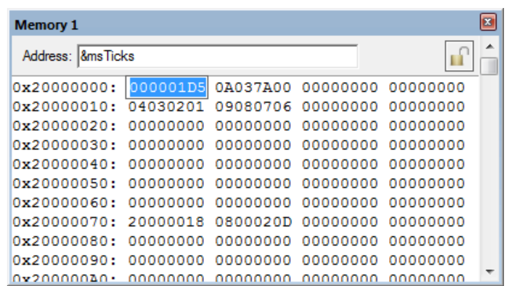
The **Register** window shows the content of the microcontroller registers.

Open the **Registers** window from the toolbar or the menu **View – Registers Window**.

You can modify the content of a register by double-clicking on the value of a register, or pressing **F2** to edit the selected value. Currently modified registers are highlighted in blue. The window updates the values when program execution halts.

Memory Window

Monitor memory areas using **Memory Windows**.



Open a **Memory** window from the toolbar or the menu using **View – Memory Windows**.

- Enter an expression in the **Address** field to monitor the memory area.
- To modify memory content, use the **Modify Memory at …** command from context menu of the **Memory** window double-click on the value.
- The **Context Menu** allows you to select the output format.
- To update the **Memory Window** periodically, enable **View – Periodic Window Update**. Use **Update Windows** in the **Toolbox** to refresh the windows manually.

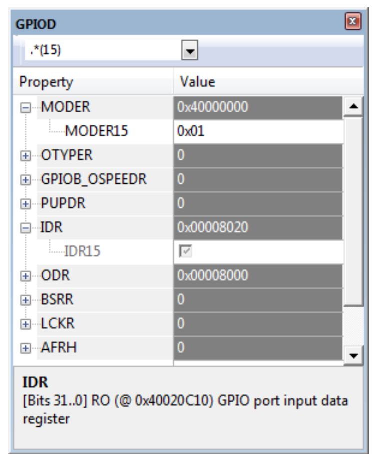
Stop refreshing the **Memory** window by clicking the **Lock** button. You can use the Lock feature to compare values of the same address space by viewing the same section in a second **Memory** window.

Peripheral Registers

Peripheral registers are memory mapped registers to which a processor can write to and read from to control a peripheral. The menu **Peripherals** provides access to **Core Peripherals**, such as the Nested Vector Interrupt Controller or the System Tick Timer. You can access device peripheral registers using the **System Viewer**.

NOTE
The content of the menu Peripherals changes with the selected microcontroller.

System Viewer



System Viewer windows display information about device peripheral registers.

Open a peripheral register from the toolbar or the menu **Peripherals – System Viewer**.

With the **System Viewer,** you can:

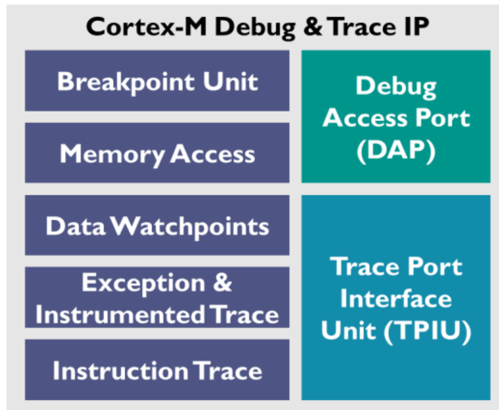
- View peripheral register properties and values. Values are updated periodically when **View — Periodic Window Update** is enabled.
- Change property values while debugging.
- Search for specific properties using **TR1 Regular Expressions** in the search field. The appendix of the ***µVision User's Guide*** describes the syntax of regular expressions.

For details about accessing and using peripheral registers, refer to the online documentation.

Trace

Run-Stop Debugging, as described previously, has some limitations that become apparent when testing time-critical programs, such as motor control or complex communication applications. As an example, breakpoints and single stepping commands change the dynamic behavior of the system. As an alternative, use the trace features explained in this section to analyze running systems.

Cortex-M processors integrate CoreSight logic that is able to generate the following trace information using:



- **Data Watchpoints** record memory accesses with data value and program address and, optionally, stop program execution.
- **Exception Trace** outputs details about interrupts and exceptions.
- **Instrumented Trace** communicates program events and enables printf-style debug messages and the RTOS Event Viewer.
- **Instruction Trace** streams the complete program execution for recording and analysis.

The **Trace Port Interface Unit** (TPIU) is available on most Cortex-M3, Cortex-M4, and Cortex-M7 processor-based microcontrollers and outputs above trace information via:

- **Serial Wire Trace Output** (SWO) works only in combination with the Serial Wire Debug mode (not with JTAG) and does not support Instruction Trace.
- **4-Pin Trace Output** is available on high-end microcontrollers and has the high bandwidth required for Instruction Trace.

On some microcontrollers, the trace information can be stored in an on-chip **Trace Buffer** that can be read using the standard debug interface.

- Cortex-M3, Cortex-M4, and Cortex-M7 has an optional **Embedded Trace Buffer** (ETB) that stores all trace data described above.
- Cortex-M0+ has an optional **Micro Trace Buffer** (MTB) that supports Instruction Trace only.

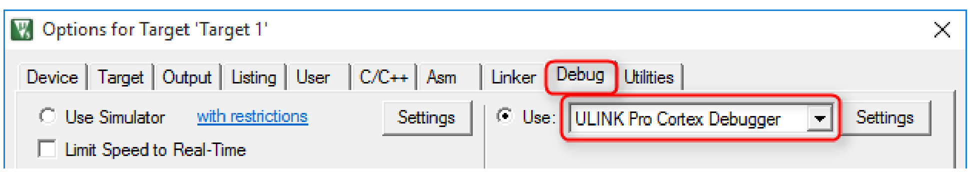
The required trace interface needs to be supported by both the microcontroller and the debug adapter. The following table shows supported trace methods of various debug adapters.

Feature	ULINK*pro*	ULINK*pro*-D	ULINK2	ST-Link v2
Serial Wire Output (SWO)	✓	✓	✓	✓
Maximum SWO clock frequency	200 MHz	200 MHz	3.75 MHz	2 MHz
4-Pin Trace Output for Streaming	✓	✗	✗	✗
Embedded Trace Buffer (ETB)	✓	✓	✓	✗
Micro Trace Buffer (MTB)	✓	✓	✓	✗

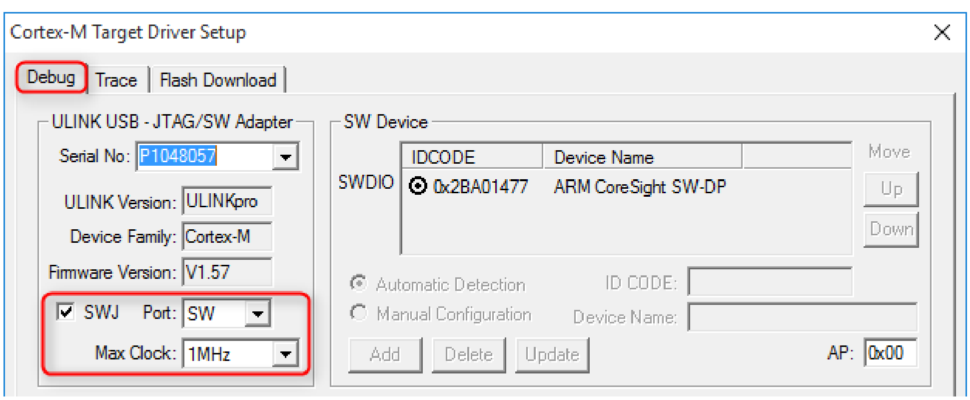
Trace with Serial Wire Output

To use the Serial Wire Trace Output (SWO), use the following steps:

Click **Options for Target** on the toolbar and select the **Debug** tab. Verify that you have selected and enabled the correct *debug adapter*.

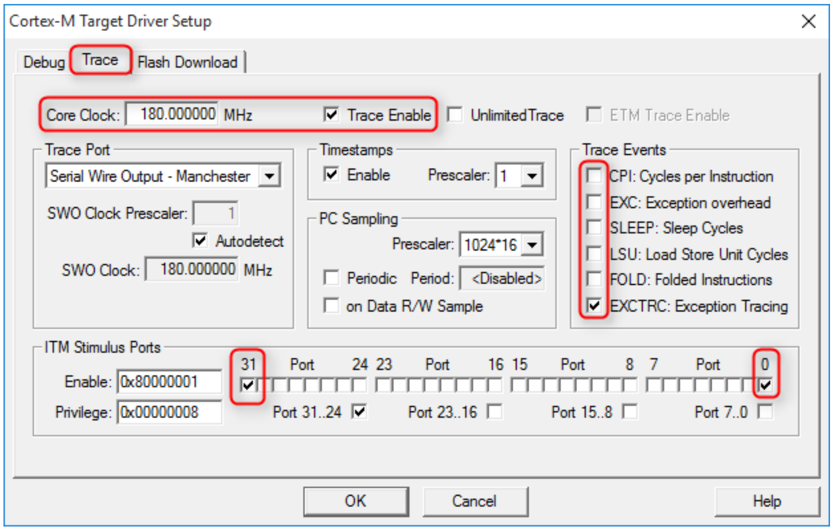


Click the **Settings** button. In the **Debug** dialog, select the debug **Port: *SW*** and set the **Max Clock** frequency for communicating with the debug unit of the device.



☞ Click the **Trace** tab. Ensure the **Core Clock** has the right setting. Set **Trace Enable** and select the **Trace Events** you want to monitor.

- Enable **ITM Stimulus Port 0** for `printf`-style debugging.
- Enable **ITM Stimulus Port 31** to view RTOS Events.



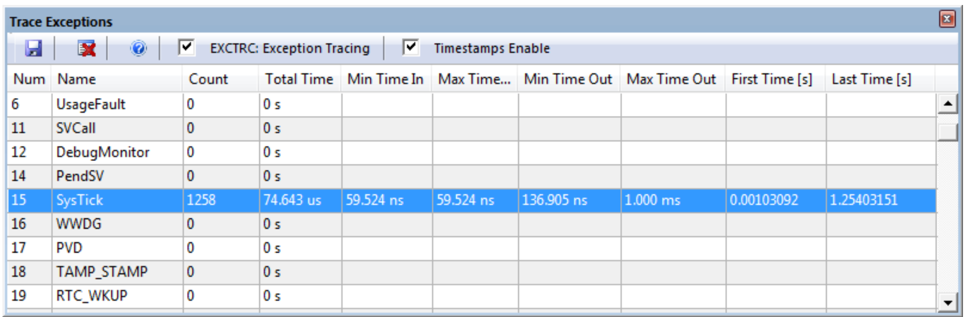
NOTE
When many trace features are enabled, the Serial Wire Output communication can overflow. The µVision Status Bar displays such connection errors.

The ULINKpro debug/trace adapter has high trace bandwidth and such communication overflows are rare. Enable only the trace features that are currently required to avoid overflows in the trace communication.

Trace Exceptions

The **Exception Trace** window displays statistical data about exceptions and interrupts.

Click on **Trace Windows** and select **Trace Exceptions** from the toolbar or use the menu **View – Trace – Trace Exceptions** to open the window.



Num	Name	Count	Total Time	Min Time In	Max Time...	Min Time Out	Max Time Out	First Time [s]	Last Time [s]
6	UsageFault	0	0 s						
11	SVCall	0	0 s						
12	DebugMonitor	0	0 s						
14	PendSV	0	0 s						
15	SysTick	1258	74.643 us	59.524 ns	59.524 ns	136.905 ns	1.000 ms	0.00103092	1.25403151
16	WWDG	0	0 s						
17	PVD	0	0 s						
18	TAMP_STAMP	0	0 s						
19	RTC_WKUP	0	0 s						

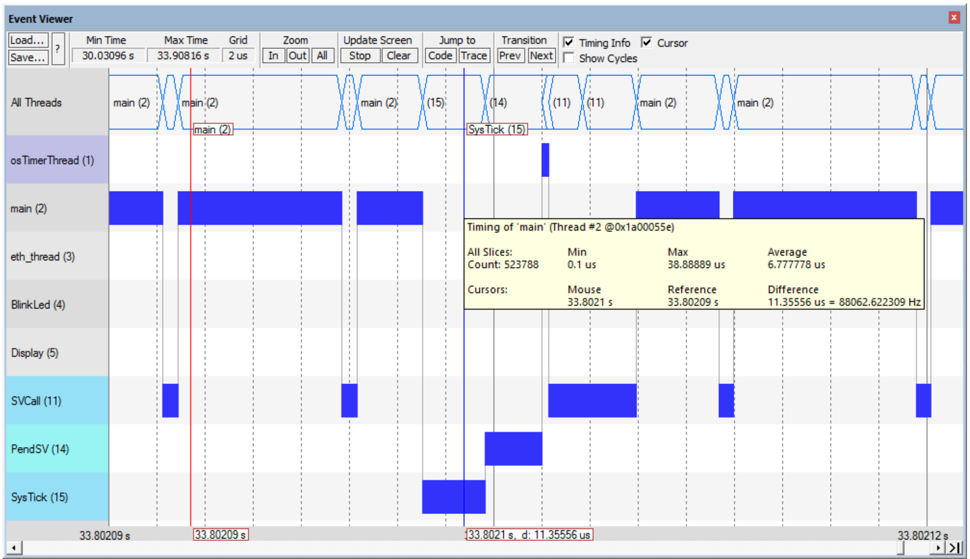
To retrieve data in the **Trace Exceptions** window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.
- Enable **EXCTRC: Exception Tracing**.
- Set **Timestamps Enable**.

NOTE
The variable accesses configured in the Logic Analyzer are also shown in the Trace Data Window.

Event Viewer

The **Event Viewer** shows RTOS thread as well as interrupt and exception timing information. Open this window with the menu **Debug – OS Support – Event Viewer**.



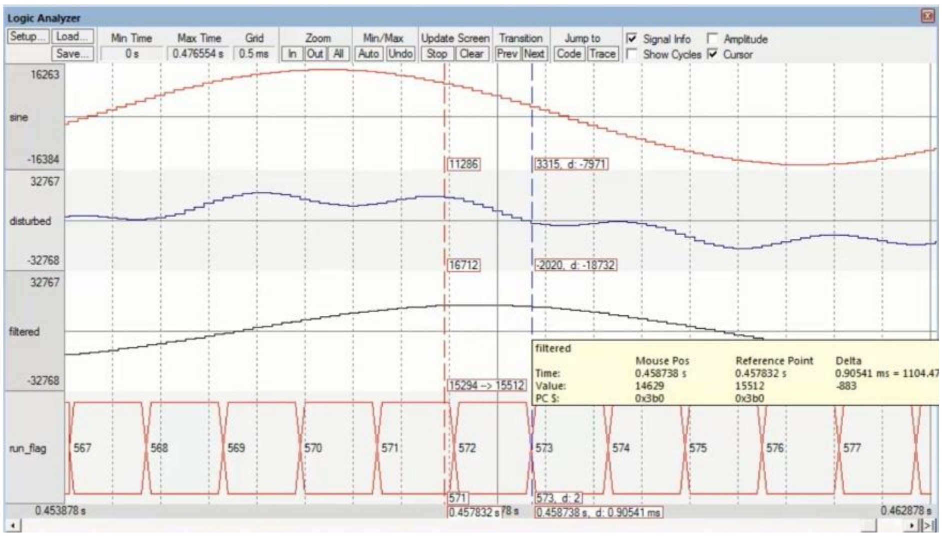
To retrieve data in the **Event Viewer** window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.
- Enable **ITM Stimulus Port 31** for CMSIS-RTOS thread timing information.
- Enable **EXCTRC: Exception Tracing** for interrupt and exception timing information.
- Set Timestamps Enable.

NOTE
The debugger provides also detailed RTOS and Thread status information that is available without Trace. Refer to ***CMSIS-RTOS System and Thread Viewer*** *on* *page 36* *for more information.*

Logic Analyzer

The Logic Analyzer window displays changes of up to four variable values over time. To add a variable to the Logic Analyzer, right click it in while in debug mode and select **Add <variable> to… - Logic Analyzer**. Open the Logic Analyzer window by choosing **View - Analysis Windows - Logic Analyzer**.



To retrieve data in the **Logic Analyzer** window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.
- Set Timestamps Enable.

NOTE

The variable accesses monitored in the Logic Analyzer are also shown in the Trace Data Window. Refer to the ***µVision User's Guide – Debugging*** *for more information.*

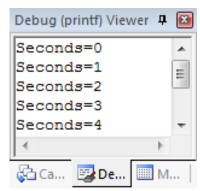
Debug (printf) Viewer

The **Debug (printf) Viewer** window displays data streams that are transmitted sequentially through the **ITM Stimulus Port 0**. To enable *printf()* debugging**,** use the **Compiler** software component as described on page 42.

This *fputc()* function redirects any *printf()* messages (as shown below) to the **Debug (printf) Viewer**.

```
int seconds;                              // Second counter
  :
while (1) {
  LED_On ();                              // Switch on
  delay ();                               // Delay
  LED_Off ();                             // Switch off
  delay ();                               // Delay
  printf ("Seconds=%d\n", seconds++);     // Debug output
}
```

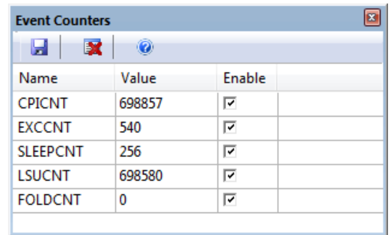
Click on **Serial Windows** and select **Debug (printf) Viewer** from the toolbar or use the menu **View – Serial Windows – Debug (printf) Viewer** to open the window.



To retrieve data in the **Debug (printf) Viewer** window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.
- Set Timestamps Enable.
- Enable ITM Stimulus Port **0**.

Event Counters



Event Counters displays cumulative numbers, which show how often an event is triggered.

From toolbar use **Trace Windows – Event Counters**

From menu **View – Trace – Event Counters**

To retrieve data in this window:

- Set **Trace Enable** in the Debug Settings Trace dialog as described above.
- Enable **Event Counters** as needed in the dialog.

Event counters are performance indicators:

- **CPICNT**: Exception overhead cycle: indicates Flash wait states.
- **EXCCNT**: Extra Cycle per Instruction: indicates exception frequency.
- **SLEEPCNT**: Sleep Cycle: indicates the time spend in sleep mode.
- **LSUCNT**: Load Store Unit Cycle: indicates additional cycles required to execute a multi-cycle load-store instruction.
- **FOLDCNT**: Folded Instructions: indicates instructions that execute in zero cycles.

Trace with 4-Pin Output

Using the 4-pin trace output provides all the features described in the section **Trace with Serial Wire Output**, but has a higher trace communication bandwidth. Instruction trace is also possible.

The **ULINKpro debug/trace adapter** supports this parallel 4-pin trace output (also called ETM Trace) which gives detailed insight into program execution.

NOTE
Refer to the ***µVision User's Guide – Debugging*** *for more information about the features described below.*

When used with ULINKpro, MDK can stream the instruction trace data for the following advanced analysis features:

- **Code Coverage** marks code that has been executed and gives statistics on code execution. This helps to identify sporadic execution errors and is frequently a requirement for software certification.
- The **Performance Analyzer** records and displays execution times for functions and program blocks. It shows the processor cycle usage and enables you to find hotspots in algorithms for optimization.
- The **Trace Data Window** shows the history of executed instructions for Cortex-M devices.

Trace with On-Chip Trace Buffer

In some cases, trace output pins are no available on the microcontroller or target hardware. As an alternative, an on-chip **Trace Buffer** can be used that supports the **Trace Data Window**.

Middleware

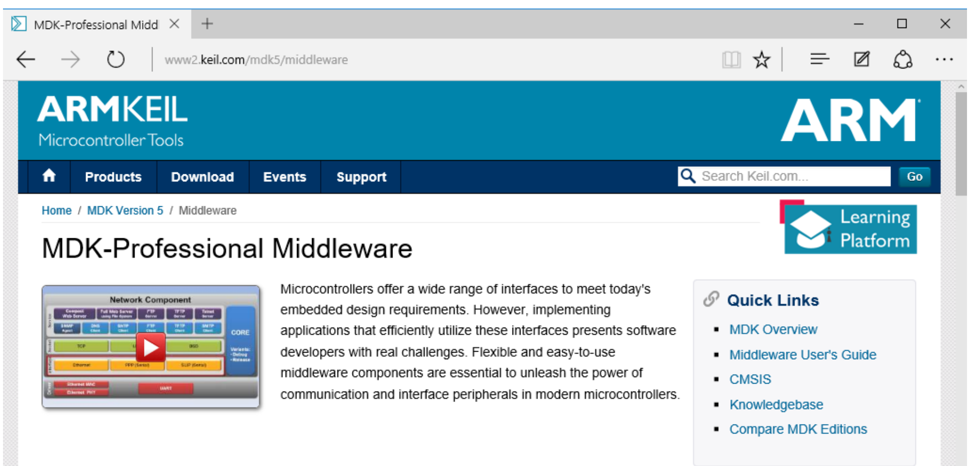
Today's microcontroller devices offer a wide range of communication peripherals to meet many embedded design requirements. Middleware is essential to make efficient use of these complex on-chip peripherals.

NOTE
This chapter describes the middleware that is part of MDK-Professional and MDK-Plus. MDK also works with middleware available from several other vendors.

Refer to ***http://www.keil.com/pack*** *for a list of public Software Packs.*

The **MDK-Middleware** Software Pack includes royalty-free middleware with components for **TCP/IP networking**, **USB Host** and **USB Device** communication, **file system** for data storage, and a **graphical user interface**.

Refer to **www.keil.com/middleware** for more information.



This web page provides an overview of the middleware and links to:

- MDK Middleware **User's Guide**
- **Device List** along with information about device-specific drivers
- Information about **Example Projects** with usage instructions

The middleware interfaces to the device peripherals using device-specific CMSIS-Drivers. Refer to **CMSIS-Driver** on page 39 for more information.

Combining several components is common for a microcontroller application. The **Manage Run-Time Environment** dialog makes it easy to select and combine MDK Middleware. It is even possible to expand the middleware component list with third-party components that are supplied as a Software Pack.

Typical examples for the usage of MDK Middleware are:

- Web server with storage capabilities: Network and File System Component
- USB memory stick: USB Device and File System Component
- Industrial control unit with display and logging functionality: Graphics, USB Host, and File System Component

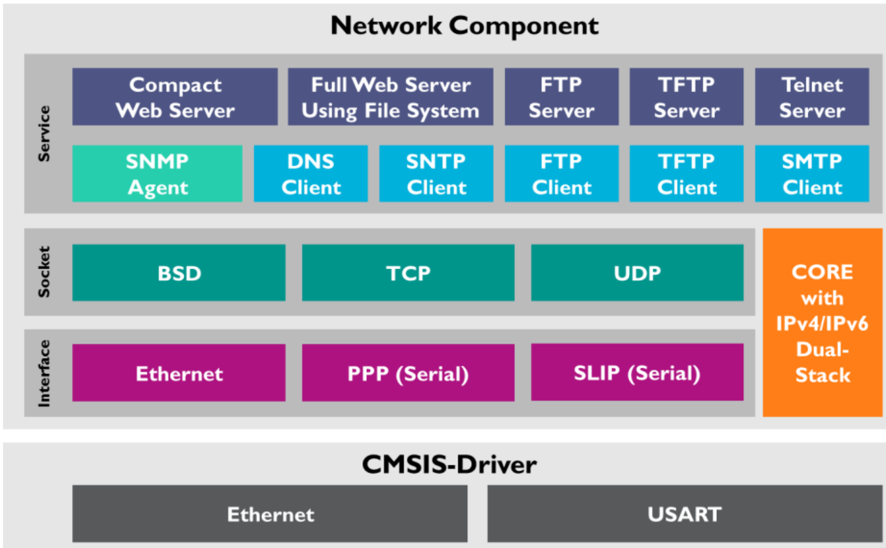
Refer to the **FTP Server Example** on page 89 that exemplifies a combination of several middleware components.

The following sections give an overview for each software component of the MDK Middleware.

NOTE
A seven days evaluation license for MDK-Professional is delivered with each installation. Refer to the ***Installation*** *chapter on page 9 for more information.*

Network Component

The **Network Component** uses TCP/IP communication protocols and contains support for services, protocol sockets, and physical communication interfaces. It supports IPv4 and IPv6 connections.



The various **services** provide program templates for common networking tasks.

- **Compact Web Server** stores web pages in ROM whereas the **Full Web Server** uses the **File System** component for page data storage. Both servers support dynamic page content using CGI scripting, AJAX, and SOAP technologies.
- **FTP** or **TFTP** support file transfer. FTP provides full file manipulation commands, whereas TFTP can boot load remote devices. Both are available for the client and server.
- **Telnet Server** provides a command line interface over an IP network.
- **SNMP Agent** reports device information to a network manager using the Simple Network Management Protocol.
- **DNS Client** resolves domain names to the respective IP address. It makes use of a freely configurable name server.
- **SNTP Client** synchronizes clocks and enables a device to get an accurate time signal over the data network.
- **SMTP Client** sends status emails using the Simple Mail Transfer Protocol.

All **Services** rely on a communication socket that can be either **TCP** (a connection-oriented, reliable full-duplex protocol), **UDP** (transaction-oriented protocol for data streaming), or **BSD** (Berkeley Sockets interface).

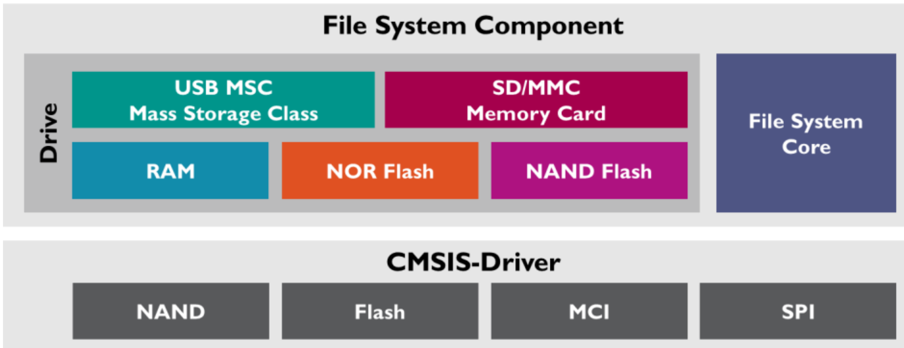
The physical **interface** can be either **Ethernet** (for LAN connections) or a serial connection such as **PPP** (for a direct connection between two devices) or **SLIP** (Internet Protocol over a serial connection).

Depending on the interface, the Network Component relies on a **CMSIS-Driver** to be present for providing the device-specific hardware interface. Ethernet requires an **Ethernet MAC** and **PHY** driver, whereas serial connections (PPP/SLIP) require a **UART** or a **Modem** driver.

The **Network Core** is available in a *Debug* variant with extensive diagnostic messages and a *Release* variant that omits these diagnostics. It supports IP communication using IPv4 and IPv6.

File System Component

The **File System Component** allows your embedded applications to create, save, read, and modify files in storage devices such as RAM, NAND or NOR Flash, memory cards, or USB memory sticks.



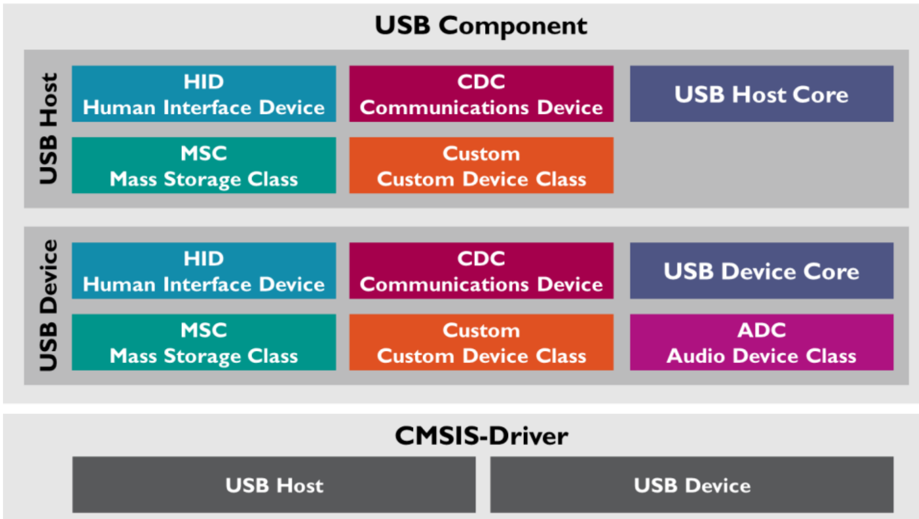
Each storage device is accessed and referenced as a **Drive**. The File System Component supports multiple drives of the same type. For example, you might have more than one memory card in your system.

The **File System Core** is thread-safe, supports simultaneous access to multiple drives, and uses a FAT system available in two file name variants: short 8.3 file names and long file names with up to 255 characters.

To access the physical media, for example NAND and NOR Flash chips, or memory cards using MCI or SPI, **CMSIS-Driver** have to be present.

USB Component

The **USB Device component** implements USB Host and Device functionality and uses standard device driver classes that are available on most computer systems, avoiding host driver development.



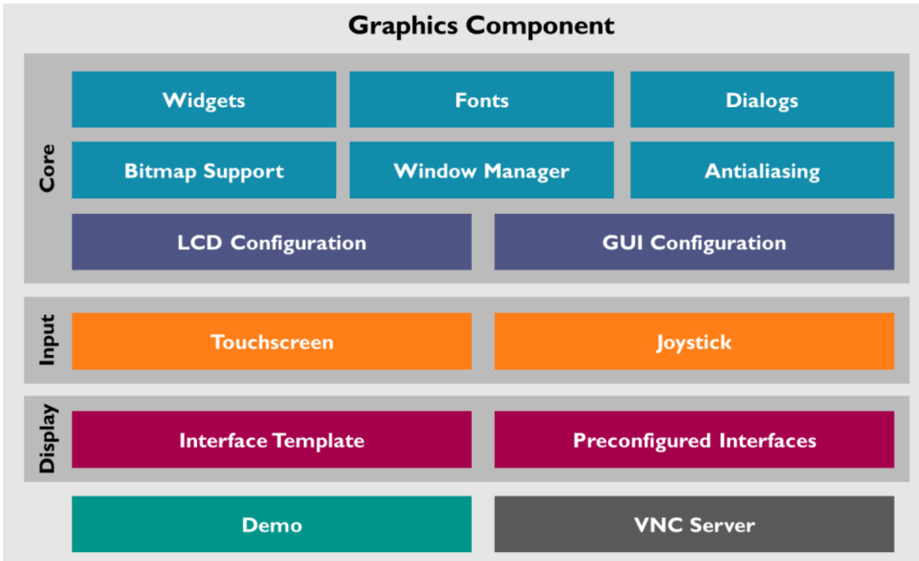
- **Human Interface Device Class** (HID) implements a keyboard, joystick or mouse. However, HID can also be used for simple data exchange.
- Use the **Mass Storage Class** (MSC) for file exchange (for example a USB memory stick).
- **Communication Device Class** (CDC) implements a virtual serial port (using the sub-class ACM) or a network connection (using the sub-class NCM).
- **Audio Device Class** (ADC) performs audio streaming.
- Use the **Custom Class** for new or unsupported USB classes.

The USB Component supports **Composite USB devices** that implement multiple device classes.

This component requires a **USB CMSIS-Driver** to be present. Depending on the application, it has to comply with the USB 1.1 (Full-Speed USB) and/or the USB 2.0 (High-Speed USB) specification.

Graphics Component

The **Graphics Component** is a comprehensive library that includes everything you need to build graphical user interfaces.



Core functions include:

- A **Window Manager** to manipulate any number of windows or dialogs.
- Ready-to-use **Fonts** and window elements**,** called **Widgets**, and **Dialogs.**
- **Bitmap Support** including JPEG and other common formats.
- **Anti-Aliasing** for smooth display.
- Flexible, configurable **Display** and **User Interface** parameters.
- The user interface can be controlled using input devices like a **Touch Screen** or a **Joystick**.

The Graphics Component interfaces to a wide range of display controllers using **preconfigured interfaces** for popular displays. Adapt the **interface template** to add support for new displays.

The **VNC Server** allows remote control of your graphical user interface via TCP/IP using the **Network Component**.

Demo shows all main features and is a rich source of code snippets for the GUI.

IoT Connectivity

The middleware in MDK-Professional provides interfaces to mbed software components that enable secure communication and Internet of Things (IoT) connectivity.

- **mbed TLS** adds cryptographic and SSL/TLS capabilities with a library collection optimized for embedded systems.
- **mbed Client** implements the OMA Lightweight M2M protocol (from Open Mobile Alliance **http://openmobilealliance.org**) and interfaces to the mbed Device Server that connects IoT devices to web applications.

Migrating to Middleware Version 7

MDK has built-in features that help you to migrate your µVision projects to the new Middleware Version 7. Most components only require a configuration file update (see below). However, the Network Component requires more migration work as it has changed from IPv4-only to dual-stack support for IPv4/IPv6.

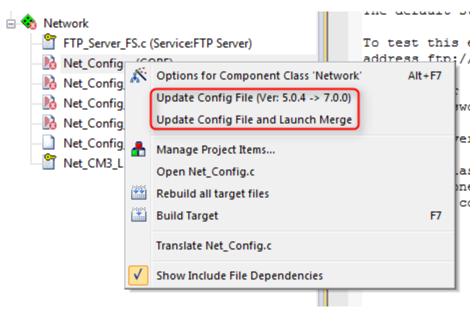
Network Component Changes

Core Changes

The Network Component's Core was previously available in a **Release** or **Debug** variant. In Middleware Version 7 this is changed to **IPv4/IPv6 Release** or **IPv4/IPv6 Debug.** When you open a project with the old component, you will see an error in the **Build Output** window. Please change to the corresponding new variant.

Configuration File Update

Special icons in the **Project** window of µVision highlight configuration files that require an update. You have the option either to overwrite the old configuration file or to update and merge the contents:



Go to **Tools → Configure Merge Tool** to specify the merge tool of your choice.

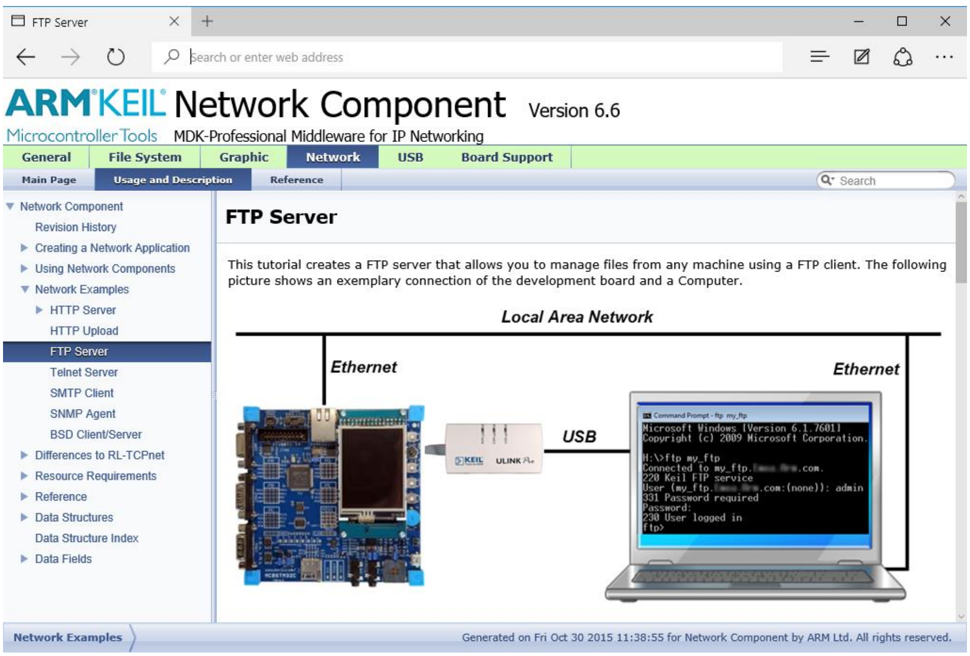
API Changes

The Network Component's documentation offers sections on how to migrate projects from the old to the new API. It offers general recommendations on the migration of services, sockets, and interfaces, as well as a side-by-side comparison of the API whether you are migrating from Middleware v5/v6 or even RL-TCPnet.

FTP Server Example

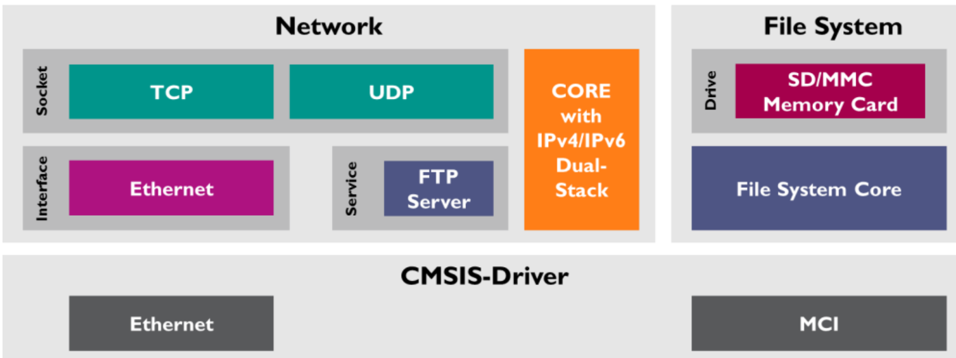
The FTP server example is a reference application that shows a combination of several middleware components. Refer to **Verify Installation using Example Projects** on page 12 for more information on the various example projects that are available.

When using an FTP Server, you can exchange and manipulate files over a TCP/IP network. The middleware documentation has more details about the FTP Server and the reference application:



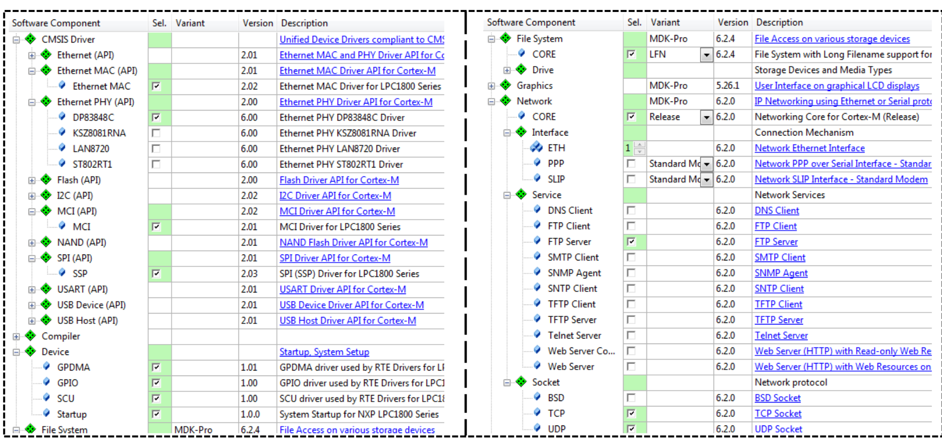
Several middleware components are the building blocks of this FTP server. A **File System** is required to handle the file manipulation. Various parts of the **Network** component build up the networking interface.

The following software components from the MDK Middleware are required to create the FTP Server example:



As explained before, CMSIS-Driver provides the interface between the microcontroller peripherals and the MDK Middleware.

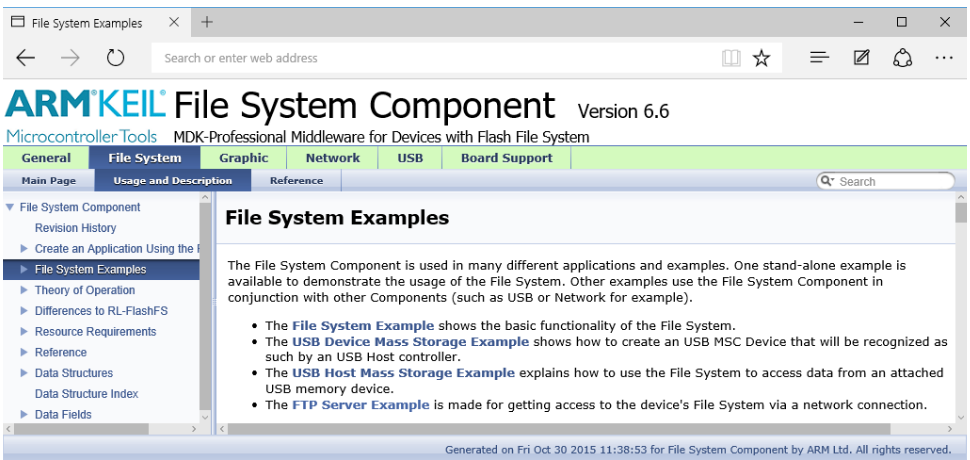
The **Manage Run-Time Environment** dialog shows the software components selected for the FTP Server example:



Using Middleware

Create your own applications using MDK Middleware components. For more information, refer to the MDK Middleware User's Guide that has sections for every component describing:

- **Example projects** outline key product features of software components. The examples are tested, implemented, and proven on several evaluation boards. Use them as reference applications or a starting point for your development.
- **Resource Requirements** describe the thread and stack resources for CMSIS-RTOS and the memory footprint.
- **Create an Application** contains the required steps for using the components in an embedded application.
- **Reference** contains the API and file documentation.



The learning platform www.keil.com/learn offers several tutorials and videos that exemplify typical use cases of the middleware. Refer also to these application notes:

- USB Host Application with File System and Graphical User Interface: www.keil.com/appnotes/docs/apnt_268.asp
- Web-Enabled MEMS Sensor Platform: www.keil.com/appnotes/docs/apnt_271.asp
- Web-Enabled Voice Recorder: www.keil.com/appnotes/docs/apnt_272.asp
- Analog/Digital Data Logger with USB Device Interface: www.keil.com/appnotes/docs/apnt_273.asp

The generic steps to use the various middleware components are:

- **Add Software Components** (page 94): In the **Manage Run-Time Environment** dialog select the software components that are required for your application.
- **Configure Middleware** (page 96): Adjust the parameters of the software components in the related configuration files.
- **Configure Drivers** (page 98): Identify and configure the peripheral interfaces that connect the middleware components to physical I/O pins of the microcontroller.
- **Adjust System Resources** (page 99): The middleware components use RTOS, memory, and stack resources and this may imply configurations, for example to CMSIS-RTOS RTX.
- **Implement Application Features** (page 100): Use the API functions of the selected components to implement the application specific behaviour. Code templates help you to create the related source code.
- **Build and Download** (page 103): After compiling and linking of the application use the steps described in the chapter **Using the Debugger** on page 64 to download the image to your target hardware.
- **Verify and Debug** (page 103): Test utilities along with debug and trace features are described in the chapter Create Applications (page 45).

USB Device HID Example

While above steps are generic and apply to all components of the MDK Middleware, the following USB Device HID example shows these steps in practice. This example creates a USB HID Device application that connects a microcontroller to a host computer via USB. On the PC the utility program ***HIDClient.exe*** is used to control LEDs on the development board.

This USB Device HID example uses the MCB1800 development board populated with a LPC1857 microcontroller. It is based on the project **Blinky with CMSIS-RTOS RTX** on page 45 along with the source files *main.c*, *LED.c*, *LED.h*, and the configuration files.

NOTE
You must adapt the code and pin configurations when using this example on other starter kits or evaluation boards. This example is available as a pre-built project in Pack Installer for many microcontroller device families supporting CMSIS-Driver.

Add Software Components

To create the USB Device HID example, start with the project **Blinky with CMSIS-RTOS RTX** described on page 45.

Use the **Manage Run-Time Environment** dialog to add specific software components.

From *USB Component* (described on page 85**):**

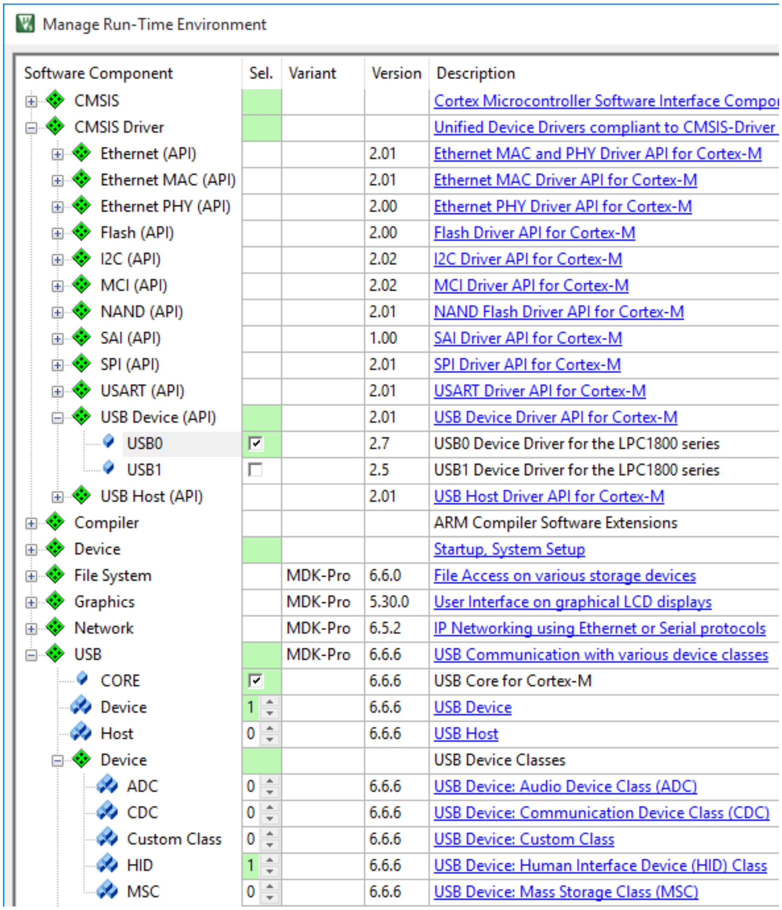
- Select **::USB:CORE** to include the basic functionality required for USB communication.
- Set **::USB:Device** to '1' to create one USB Device instance.
- Set **::USB:Device:HID** to '1' to create a HID Device Class instance. If you select multiple instances of the same class or include other device classes, you will create a Composite USB Device.

From *CMSIS-Driver* (described on page 39**):**

Select from **::CMSIS Driver:USB Device (API)** an appropriate driver suitable for your application. Some devices may have specific drivers for USB Full-Speed and High-Speed whereas other microcontrollers may have a combined driver. Here, select **USB0**.

`TIP:` Click on the hyperlinks in the **Description** column to view detailed documentation for each software component.

The picture below shows the **Manage Run-Time Environment** dialog after adding these components.


Manage Run-Time Environment

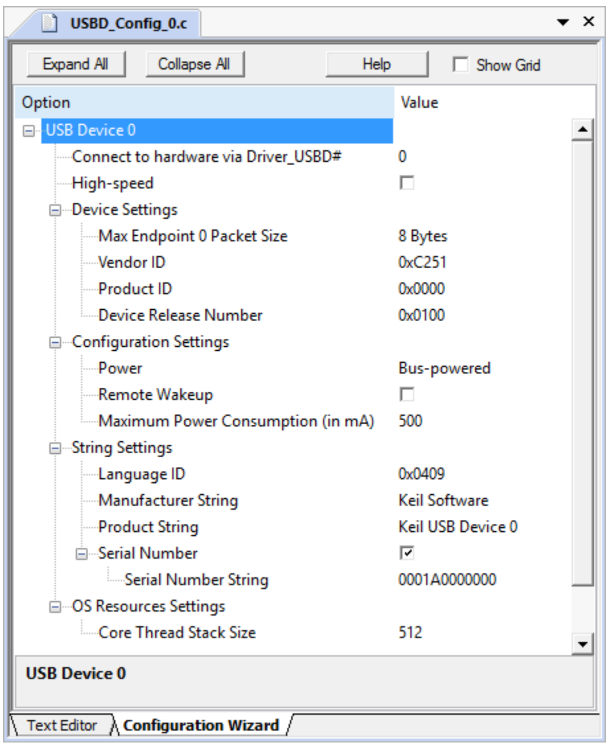
Software Component	Sel.	Variant	Version	Description
CMSIS				Cortex Microcontroller Software Interface Compo
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver
Ethernet (API)			2.01	Ethernet MAC and PHY Driver API for Cortex-M
Ethernet MAC (API)			2.01	Ethernet MAC Driver API for Cortex-M
Ethernet PHY (API)			2.00	Ethernet PHY Driver API for Cortex-M
Flash (API)			2.00	Flash Driver API for Cortex-M
I2C (API)			2.02	I2C Driver API for Cortex-M
MCI (API)			2.02	MCI Driver API for Cortex-M
NAND (API)			2.01	NAND Flash Driver API for Cortex-M
SAI (API)			1.00	SAI Driver API for Cortex-M
SPI (API)			2.01	SPI Driver API for Cortex-M
USART (API)			2.01	USART Driver API for Cortex-M
USB Device (API)			2.01	USB Device Driver API for Cortex-M
USB0	☑		2.7	USB0 Device Driver for the LPC1800 series
USB1	☐		2.5	USB1 Device Driver for the LPC1800 series
USB Host (API)			2.01	USB Host Driver API for Cortex-M
Compiler				ARM Compiler Software Extensions
Device				Startup, System Setup
File System		MDK-Pro	6.6.0	File Access on various storage devices
Graphics		MDK-Pro	5.30.0	User Interface on graphical LCD displays
Network		MDK-Pro	6.5.2	IP Networking using Ethernet or Serial protocols
USB		MDK-Pro	6.6.6	USB Communication with various device classes
CORE	☑		6.6.6	USB Core for Cortex-M
Device	1		6.6.6	USB Device
Host	0		6.6.6	USB Host
Device				USB Device Classes
ADC	0		6.6.6	USB Device: Audio Device Class (ADC)
CDC	0		6.6.6	USB Device: Communication Device Class (CDC)
Custom Class	0		6.6.6	USB Device: Custom Class
HID	1		6.6.6	USB Device: Human Interface Device (HID) Class
MSC	0		6.6.6	USB Device: Mass Storage Class (MSC)

Configure Middleware

Every MDK Middleware component has a set of configuration files that adjusts application specific parameters and determines the driver interfaces. Access these configuration files from the **Project** window in the component class group. They usually have names like *<Component>_Config_0.c* or *<Component>_Config_0.h*.

Some of the settings in these files require corresponding settings in the driver and device configuration file (*RTE_Device.h*) that is subject of the next section.

For the USB HID Device example, there are two configuration files available: *USBD_Config_0.c* and *USBD_Config_HID_0.h*.



The file *USBD_Config_0.c* contains a number of important settings for the specific USB Device:

- The setting **Connect to Hardware via Driver_USBD#** specifies the *control struct* that reflects the peripheral interface, in this case, the USB controller used as device interface. For microcontrollers with only one USB controller the number is ‘0’. Refer to CMSIS-Driver on page 39 for more information.
- Select **High-Speed** if supported by the USB controller. Using this setting requires a driver that supports USB High-Speed communication.
- Set the **Vendor ID (VID)** to a private VID. The USB Implementer’s Forum **http://www.usb.org/developers/vendor** provides more information on how to apply for a valid vendor ID.
- Every device needs a unique **Product ID**. The host computer's operating system uses it together with the VID to find a suitable driver for your device.
- Set the **Manufacturer** and the **Product String** to identify the USB device in PC operating systems.

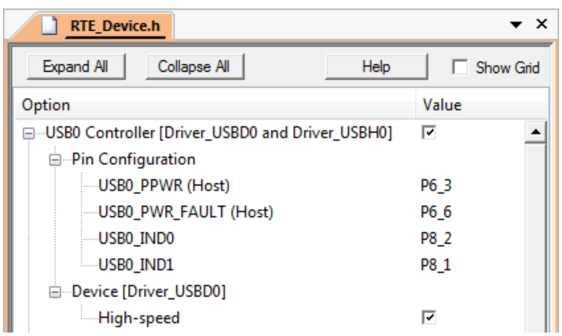
The file *USBD_Config_HID_0.h* contains device class specific Endpoint settings. For this example, no changes are required.

Configure Drivers

Drivers have certain properties that define attributes such as I/O pin assignments, clock configuration, or usage of DMA channels. For many devices, the *RTE_Device.h* configuration file contains these driver properties. It typically requires configuration of the actual peripheral interfaces used by the application. Depending on the microcontroller device, you can enable different hardware peripherals, specify pin settings, or change the clock settings for your implementation.

The USB HID Device example requires the following settings:

- Enable **USB0 Controller** and expand this section.
- Change the **Pin Configuration** as depicted below.
- Enable Device:High-speed.



Adjust System Resources

Every middleware component has certain memory and RTOS resource requirements. The section "**Resource Requirements**" in the MDK Middleware User's Guide documents the requirements for each component.

Most middleware components use the CMSIS-RTOS. It is important that the RTOS is configured to support the requirements.

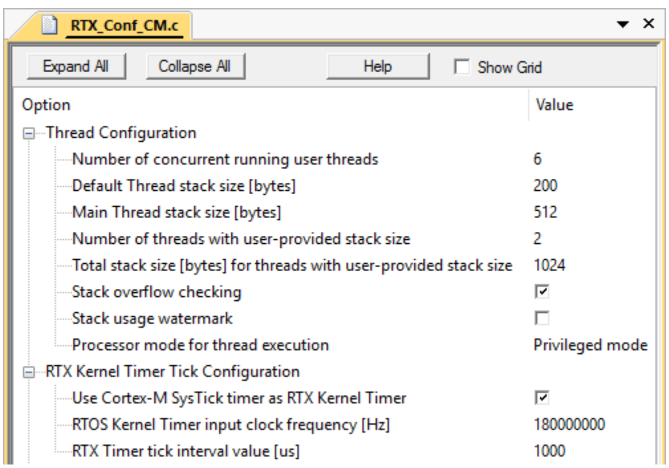
For CMSIS-RTOS RTX, the *RTX_Conf_CM.c* file configures threads and stacks settings. Refer to **CMSIS-RTOS RTX Configuration** on page 30 for more information.

For the USB HID Device example, the following settings apply:

- The **::USB:Device** component requires one thread (called USBD*n*_CoreThread) and a user-provided stack of 512 bytes.
- The **::USB:Device:HID** component also requires one thread (called USBD_HID*n*_Thread) and a user-provided stack of 512 bytes.

Reflect these requirements with the settings in the *RTX_Conf_CM.c* file:

- **Number of concurrent running threads:** 6 (default) is enough to run the two threads of the USB Device component concurrently. Adjust this setting if the user application executes additional threads.
- **Default Thread stack size [bytes]:** This setting is not important as the USB component runs on user-provided stack.
- **Main Thread stack size [bytes]:** 512. Stack is required for the API calls that initialize the USB Device component.
- Number of threads with user-provided stack size: 2. Specifies the two threads (for ::USB:Device and ::USB:Device:HID) with a user-provided stack.
- Total stack size [bytes] for threads with user-provided stack size: 1024. Specifies the total stack size of the two threads.
- The **Timer Clock value [Hz]** needs to match the system clock (180000000).

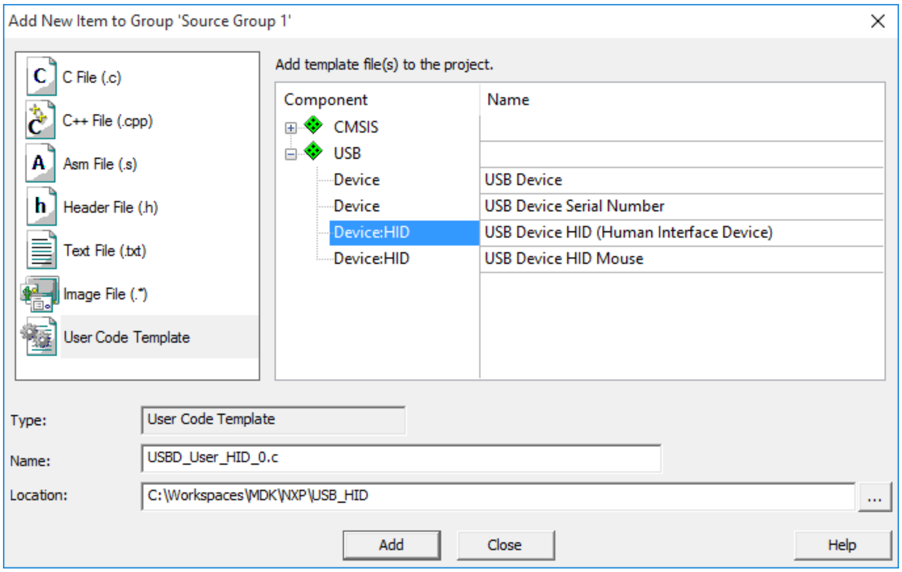


Implement Application Features

Now, create the code that implements the application specific features. This includes modifications to the files *main.c*, *LED.c*, and *LED.h* that were created initially for the project **Blinky with CMSIS-RTOS RTX** on page 45.

The middleware provides **User Code Templates** as starting point for the application software.

☞ In the **Project** window, right-click **Source Group 1** and open the dialog **Add New Item to Group**. Select the user code template from **::USB:Device:HID - USB Device HID (Human Interface Device)** and click **Add**.



To connect the PC USB application to the microcontroller device, modify the function *USBD_HID0_SetReport()*, which handles data coming from the USB Host. For this example, the data are created with the utility **HIDClient.exe**.

☞ Open the file *USBD_User_HID_0.c* in the editor and modify the code as shown below. This will control the LEDs on the evaluation board.

```
#include "LED.h"                      // access functions to LEDs
 :
bool USBD_HID0_SetReport (uint8_t rtype, uint8_t req, uint8_t rid,
                          const uint8_t *buf, int32_t len) {
  uint8_t i;

  switch (rtype) {
    case HID_REPORT_OUTPUT:
      for (i = 0; i < 4; i++) {
        if (*buf & (1 << i)) LED_On (i);
        else                 LED_Off (i);
      }
      break;

    case HID_REPORT_FEATURE:
      break;
  }
  return true;
}
```

Expand the functions in the file *LED.c* to control several LEDs on the board and remove the thread that blinks the LED, as it is no longer required.

☞ Open the file *LED.c* in the editor and modify the code as shown below.

```
/*----------------------------------------------------------------------------
 * File LED.c
 *---------------------------------------------------------------------------*/
#include "SCU_LPC18xx.h"
#include "GPIO_LPC18xx.h"
#include "cmsis_os.h"                     // ARM::CMSIS:RTOS:Keil RTX

const GPIO_ID LED_GPIO[] = {              // LED GPIO definitions
  { 6, 24 },
  { 6, 25 },
  { 6, 26 },
  { 6, 27 }
};

void LED_Initialize (void) {
  GPIO_PortClock     (1);                 // Enable GPIO clock

  /* Configure pin: Output Mode with Pull-down resistors */
  SCU_PinConfigure (13, 10, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 24, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 24, 0);
  SCU_PinConfigure (13, 11, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 25, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 25, 0);
  SCU_PinConfigure (13, 12, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 26, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 26, 0);
  SCU_PinConfigure (13, 13, (SCU_CFG_MODE_FUNC4|SCU_PIN_CFG_PULLDOWN_EN));
  GPIO_SetDir      (6, 27, GPIO_DIR_OUTPUT);
  GPIO_PinWrite    (6, 27, 0);
}

void LED_On (uint32_t num) {
  GPIO_PinWrite     (LED_GPIO[num].port, LED_GPIO[num].num, 1);
}

void LED_Off (uint32_t num) {
  GPIO_PinWrite     (LED_GPIO[num].port, LED_GPIO [num].num, 0);
}
```

☞ Open the file *LED.h* in the editor and modify it to coincide with the changes to *LED.c*. The functions *LED_On()* and *LED_Off()* now have a parameter.

```
/*----------------------------------------------------------------------------
 * File LED.h
 *---------------------------------------------------------------------------*/
void LED_Initialize ( void );
void LED_On ( uint32_t num );
void LED_Off ( uint32_t num );
```

☞ Change the file *main.c* as shown below. Instead of starting the thread that blinks the LED, add code to initialize and start the USB Device Component. Refer to the Middleware User's Guide for further details.

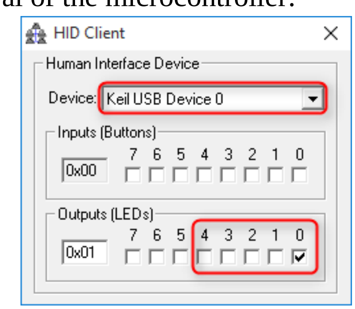
```
/*----------------------------------------------------------------------------
 * File main.c
 *---------------------------------------------------------------------------*/
#define osObjectsPublic                    // define objects in main module
#include "osObjects.h"                     // RTOS object definitions
#include "LPC18xx.h"                       // Device header
#include "LED.h"                           // Initialize and set GPIO Port
#include "rl_usb.h"                        // Keil.MDK-Pro::USB:CORE

/*
 * main: initialize and start the system
 */
int main (void) {
  osKernelInitialize ();          // Initialize CMSIS-RTOS

  // initialize peripherals here
  LED_Initialize ();              // Initialize LEDs

  USBD_Initialize (0);            // USB Device 0 Initialization
  USBD_Connect (0);               // USB Device 0 Connect

  osKernelStart ();               // Start thread execution
  while (1);
}
```

Build and Download

Build the project and download it to the target as explained in chapters **Create Applications** on page 45 and **Using the Debugger** on page 64.

Verify and Debug

Connect the development board to your PC using another USB cable. This provides the connection to the USB device peripheral of the microcontroller.



Once the board is connected, a notification appears that indicates the installation of the device driver for the USB HID Device.

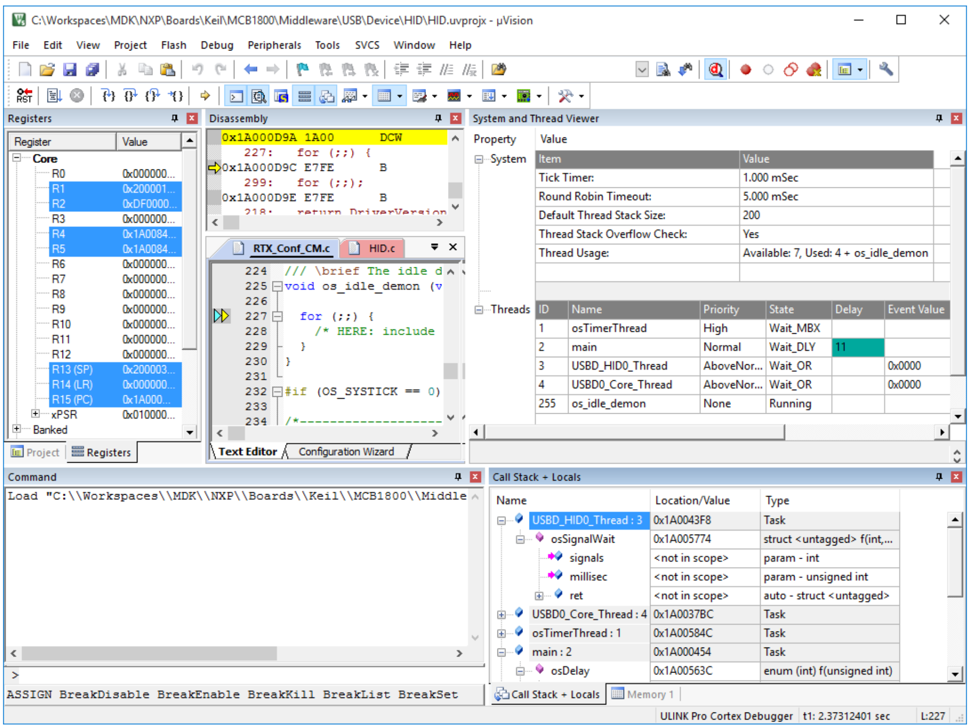
The utility program ***HIDClient.exe*** that is part of MDK enables testing of the connection between the PC and the development board. This utility is located the MDK installation folder *.\Keil\ARM\Utilities\HID_Client\Release*.

To test the functionality of the USB HID device run the ***HIDClient.exe*** utility and follow these steps:

- Select the Device to establish the communication channel. In our example, it is "*Keil USB Device 0*".
- Test the application by changing the **Outputs (LEDs)** checkboxes. The respective LEDs will switch accordingly on the development board.

If you are having problems connecting to the development board, you can use the debugger to find the root cause.

 From the toolbar, select **Start/Stop Debug Session**.



Use debug windows to narrow down the problem. Breakpoints help you to stop at certain lines of code so that you can examine the variable contents.

NOTE

Debugging of communication protocols can be difficult. When starting the debugger or using breakpoints, communication protocol timeouts may exceed making it hard to debug the application. Therefore, use breakpoints carefully.

In case that the USB communication fails, disconnect USB, reset your target hardware, run the application, and reconnect it to the PC.

Index

A

B

C

D

E

F

G

T

U

V